Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System

By

Ray A. Bittner Jr.

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

APPROVED:

_____
Peter M. Athanas, Chairman

_____   _____
A. Lynn Abbott   Nathaniel J. Davis

_____   _____
Scott F. Midkiff   Calvin J. Ribbens

January 23, 1997

Blacksburg, Virginia

Keywords: Data Flow, DSP, FPGA, Wormhole Run-Time Reconfiguration, VLSI

Wormhole Run-Time Reconfiguration: Conceptualization and VLSI Design of a High Performance Computing System

By

Ray A. Bittner Jr.

Committee Chairman:

Peter M. Athanas

Bradley Department of Electrical Engineering

(Abstract)

In the past, various approaches to the high performance numerical computing problem have been explored. Recently, researchers have begun to explore the possibilities of using Field Programmable Gate Arrays (FPGAs) to solve numerically intensive problems. FPGAs offer the possibility of customization to any given application, while not sacrificing applicability to a wide problem domain. Further, the implementation of data flow graphs directly in silicon makes FPGAs very attractive for these types of problems. Unfortunately, current FPGAs suffer from a number of inadequacies with respect to the task. They have lower transistor densities than ASIC solutions, and hence less potential computational power per unit area. Routing overhead generally makes an FPGA solution slower than an ASIC design. Bit-oriented computational units make them unnecessarily inefficient for implementing tasks that are generally word-

oriented. And finally, in large volumes, FPGAs tend to be more expensive per unit due to their lower transistor density.

To combat these problems, researchers are now exploiting the unique advantage that FPGAs exhibit over ASICs: reconfigurability. By customizing the FPGA to the task at hand, as the application executes, it is hoped that the cost-performance product of an FPGA system can be shown to be a better solution than a system implemented by a collection of custom ASICs. Such a system is called a Configurable Computing Machine (CCM). Many aspects of the design of the FPGAs available today hinder the exploration of this field.

This thesis addresses many of these problems and presents the embodiment of those solutions in the Colt CCM. By offering word grain reconfiguration and the ability to partially reconfigure at computational element resolution, the Colt can offer higher effective utilization over traditional FPGAs. Further, the majority of the pins of the Colt can be used for both normal I/O and for chip reconfiguration. This provides higher reconfiguration bandwidth contrasted with the low percentage of pins used for reconfiguration of FPGAs. Finally, Colt uses a distributed reconfiguration mechanism called Wormhole Run-Time Reconfiguration (RTR) that allows multiple data ports to simultaneously program different sections of the chip independently. Used as the primary example of Wormhole RTR in the patent application, Colt is the first system to employ this computing paradigm.

Dedication

This dissertation is dedicated to all those who have supported me throughout my education. My family has played a large part in that role. My father, mother and my sister Valerie have provided me with financial and emotional support throughout my stay here. There have also been countless little things, like helping me to cart my stuff back and forth to Cumberland endless times and all those turkey dinners at Thanksgiving. For these things I wish to thank them and I pray God's blessings on each of them.

In recent years, Intervarsity Christian Fellowship (IVCF) has played a large role in my life as I have become an active member of that organization. Through IV I have met a large number of people who are believers in God and His Son Jesus Christ and who generally know how to have a good time. Most of my free time here is now spent with that group of people and they have taught me many things about myself, the world and my place in it. My only regret now is that I did not become involved with them sooner. But, in God's perfect timing, through many of them I have now seen the love of God and His love for me. To God's people, this dissertation is dedicated as well.

I was baptized in the last year on March 31, 1996 at the 11:10 service at Blacksburg Christian Fellowship. There I told a bit of the story of how God has become a part of my life. I could not do justice to the story then, and so I can scarce hope to here in a few sentences. There have been many times when I would have quit this degree if left to my own devices, but each time God has been faithful to me and has seen me through. He has revealed things to me many and beautiful and yet sin is still prevalent in my life. How I can know God and still allow sin in

my life is a mystery, but I know that it is for that reason that Jesus died on the cross, once for all.

So though I speak of Him last, certainly my dedication to Him should be first for in this past year

He has revealed to me the sureness of my salvation as well.

Acknowledgments

I would like to thank Dr. Peter Athanas for helping me through the mundane processes of attaining this degree and also for offering insights into the problems at hand. More importantly, I would like to thank him for his patience with me and for his endurance through our tens of hours of "discussion" about the many facets of this work. But, most of all I would like to thank him for his personable nature and his open attitude towards me and his other students.

Several people have assisted me in the realization of this project. Mark "Grover" Musgrove performed most of the VLSI layout work for the Colt device, and his efforts are documented in [1]. Dr. Peter Athanas performed all of the VLSI layout work for the crossbar. Also Mark "Chewie" Cherbaka was responsible for the majority of the testing of the Colt device and some work with mapping applications to the architecture. Chewie's work is described in [2]. Finally, Tsun Han Yang designed and performed the VLSI layout for the multiplier. These efforts were all greatly appreciated and made Colt a reality. I wish to express my thanks and gratitude to each of those involved.

I would also like to thank the other members of my committee for their time and patience as this degree has slowly taken form. In some ways it was not a conventional approach and I appreciate their understanding of my situation. I would like to thank Dr. Cal Ribbens who taught

---

1 Musgrove, Mark D., "VLSI Implementation of a Run-Time Reconfigurable Custom Computing Integrated Circuit," Master's Thesis, Virginia Tech, Department of Electrical and Computer Engineering, Blacksburg, Virginia, November 7, 1996.
2 Cherbaka, Mark F., "Verification and Configuration of a Run-Time Reconfigurable Custom Computing Integrated Circuit for DSP Applications," Master's Thesis, Virginia Tech, Department of Electrical and Computer Engineering, Blacksburg, Virginia, July 1996.

me about assemblers and didn't bail out when I desperately needed an out of department committee member. I would also like to thank Dr. Nat Davis for teaching me about computer architecture and for informing me that Kai Hwang really isn't a very big man. Equal thanks go to Dr. Lynn Abbott who taught me about pattern recognition and who also served on my Master's committee. And, of course, the data port state machine would not have been possible without Dr. Scott Midkiff who taught me all I ever needed to know about digital design.

Finally, I would like to thank Loretta Estes, Pam Crewey, Bob Lineberry, Joe Blum, Dr. John Bay, Dr. Ioannis M. Besieris and the many other people of the Bradley Department of Electrical Engineering who have helped and taught me in my many years at Virginia Tech. Through an act of God they decided to grant me a Bradley Fellowship through which the burden of the Ph.D. was eased, I would like to thank them for that and Marion Via who made it possible. Lastly, I would be remiss if I didn't mention Tom Drayer who *might* get his degree before me.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Scope

The goal of this research is to develop the computational core of a hardware platform for performing high speed Digital Signal Processing (DSP). The effort is driven by the need to attain higher computational performance to meet the escalating demand for such things as wireless communication systems. The computational needs of the DSP platform are met by the solution presented here. At the same time, the solution maintains enough flexibility to be applicable to a much broader domain of computing. The computing platform will be used in a prototype communications system that will demonstrate many algorithms designed to provide higher utilization of communications channels. Many of the demands of DSP applications are also common to more general processing tasks such as numerical scientific computing, image processing and many types of simulation.

1.2 Problem Overview

The problem of high performance computation has been a field of intense research for almost a half century. While improvements in computational ability have been gained, most of these have been achieved through the development of better manufacturing processes that allow higher speed computational cores to be constructed economically. Speedup gains through architectural innovation have been modest by comparison. One of the great enigmas of computer

architecture has been scalability; the question of how to successfully apply more than a single computational element to a problem to gain speedup without greatly sacrificing utilization efficiency. Efficiency may be lost to inherently sequential sections of the algorithm and is therefore bounded by Amdahl's law [3]. Efficiency may also be lost to communications overhead between processing elements. This communications overhead can take the form of data overlap between processors that must exchange operands in a lock-step fashion, or simply long latencies in the communications network. Many of these issues are addressed by the architectural implementation proposed.

Silicon utilization is a second domain where the efficiency of available processing elements often suffers. As will be shown in Section 2.1.2, only a small percentage of the active silicon on a microprocessor of today is devoted to actual computation of data. Most of the area is used to support the Von Neumann fetch-execute cycle by either performing a form of address decoding or some type of speedup of this process. Even within the computational circuitry itself there is a loss of efficiency. During normal operation many of the processing functions of a chip lie idle while a few are in use. Some processors can make better use of available resources simultaneously, but only at the cost of additional support hardware.

Finally, a concern in all portable systems today is power consumption. This particularly applies to systems that need to be deployed in portable radios and cellular phones. Here, power is at a premium and lower power consumption translates into longer battery life and better

---

3 Amdahl, G., "Validity of Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the Spring Joint Computer Conference*, pp. 483-485, April 1967.

customer satisfaction. Another use of low power systems is in space-based systems such as communications satellites where all available power must often come from solar power panels. Low power systems are also found in autonomous systems such as sonar nets and buoys that are isolated in the ocean waiting for a local disturbance to awaken them. On a more earthly note, low power design becomes more critical as VLSI process technology increases the density of transistors on a chip. As the transistors are moved closer together, more heat per unit area is generated and the potential for melt-down increases.

CMOS technology is noted for low quiescent power dissipation. As long as a fully complementary CMOS design remains in the same state, almost no power is consumed. Here again, the high overhead of implementing a processor based on a Von Neumann architecture leads to undue waste. Since such a small percentage of the processor is actually used for computation, all the activity in the chip that is used to support that computation is consuming power that could be used for other purposes. The area consumed by the support circuitry leads to poor computational density, and the power it dissipates leads to shortened chip and battery life. The architecture proposed in this research will attempt to address these issues as well.

1.3 Proposed Solution

Increasing Flexibility

Increasing Speed

| ASIC | Programmable Logic (FPGAs) | DSP Processor | General Purpose Microprocessor |
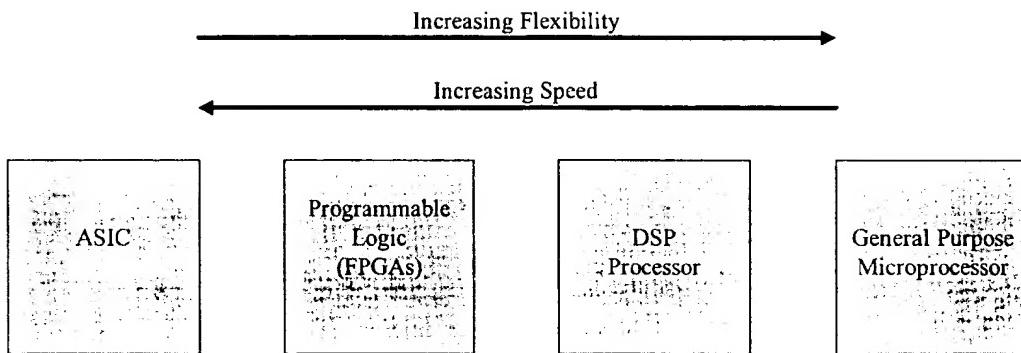
Figure 1.1 - Computing Implementation Alternatives.

Figure 1.1 is one vision of the lineup of available computing solutions today. Unquestionably, the fastest way to solve any problem is through the use of an Application Specific Integrated Circuit (ASIC), shown on the left. ASICs have a high Non-Recurring Engineering (NRE) cost initially as well as a long development cycle. These costs, coupled with the fact that silicon is an irreversible design medium, make the frequent changes during development cycles, corrections and upgrades difficult to justify. At the other end of the computing spectrum are general purpose microprocessors. These are typically based upon a Von Neumann style instruction sequence and can be quickly shifted from one computational task to the next. Unfortunately, the generality of design in these components makes them poorly suited to fit any particular computing niche. In between are Programmable Logic and Digital Signal Processors (DSPs). DSPs still execute the fetch-execute cycle, but they are geared towards a computational environment by the inclusion of extra hardware for performing fixed and floating point computations much faster. The makers of these chips can justify the extra silicon used by

4

citing the higher proportion of mathematical instructions typically present in numerically intensive applications.

Programmable Logic Devices (PLDs) bridge the gap between ASICs and microprocessors. Typified by Field Programmable Gate Arrays (FPGAs), Programmable Logic Devices are constructed in a generic manner to facilitate programming at the logic gate level after manufacture. The exact form of the function of the chip is determined by a large number of static RAM cells that connect and create whatever logic functions are required. Thus, the function performed by the chip is a simple matter of changing the values of these RAM cells. By allowing programming at such a low level, the hardware on the chip can be tailored to exactly fit the task at hand. Thus, for some applications an FPGA can outperform a DSP. However, programmable routing and associated overhead often leave FPGA performance in the wake of a well designed ASIC.

One way to improve the overall flexibility-speed product of FPGAs is to decrease the reconfiguration time, i.e. the time spent changing the value of the memory cells to perform a new task. In the past, most FPGA designs have been static in nature. A single design is loaded into the FPGA and it remains unchanged for the duration of circuit operation. Many efforts are currently underway to employ run-time reconfiguration with FPGAs to dynamically change the programming of the FPGA while the circuit is in use. The exact function of the FPGA can be targeted and re-targeted to accelerate the solution of whatever problem the system is currently trying to solve. This allows emulation of a large hardware platform using a small device. Obviously, the faster the re-targeting operation can occur, the more instances in which FPGA acceleration can be employed, and hence the faster the overall execution.

5

Existing methods for performing this reconfiguration are primitive. Typically, only a small percentage of the pins on an FPGA can be used for reconfiguring the design, thus the operation is I/O bound. Further, most FPGAs can only be reconfigured as a unit, or as blocks that represent a large percentage of the overall computational resources to be had. This limits both the degree of parallel configuration that is possible as well as fixing the degree of resource fragmentation for a given configuration. Also, while reconfiguration takes a coarse grain approach, the actual computations take a very fine grain approach, resulting in large amounts of silicon area being devoted to routing resources and configuration storage.

One of the unique contributions of this research is Wormhole Run-Time Reconfiguration (RTR). Wormhole RTR is a method for reconfiguring an FPGA, or an entire system, in an entirely distributed fashion. Routing and programming are handled at a local rather than at a global level. Without the need for global control, the amount of hardware necessary to control the configuration process on-chip diminishes greatly; thus it is possible to practically achieve finer grain divisions between computational resources and limit resource fragmentation. Further, because Wormhole RTR is a distributed paradigm, it allows different parts of the same resource, an FPGA for example, to be independently configured through many different data paths simultaneously. While care must be taken to ensure that resource allocations do not overlap, multiple independent configuration paths greatly increase the configuration bandwidth of a given device, enhancing reconfiguration speed and overall system performance. The paradigm also enhances the fault tolerance of a device. The lack of a central controller means that there are fewer single point failures that can lead to total system failure. Also, when a failure does occur, the local nature of Wormhole RTR allows it to route around the problem area and use other

6

resources that are functional. This has possibilities not only for faults that occur over a period of time, but also for faults in parts that are fresh off the assembly line, improving chip yields through dynamic reconfiguration around manufacturing defects. These advantages and more will be further expounded upon later.

The Colt chip embodies the Wormhole RTR concept. Fitting somewhere between an FPGA and a DSP in the computational spectrum, the Colt contains elements of both. Nearly all of the pins on the device serve both as configuration as well as data I/O resources, providing six independent paths onto the chip that can simultaneously be used to configure computational resources. Further, the resources on the chip are word-oriented rather than bit-oriented, giving several advantages. First, the chip requires less routing control, giving better overall propagation delay times and allowing faster operation than a bit-oriented FPGA. Second, fewer memory storage elements are required for configuration information since all bits in a given word are configured *en masse*, as opposed to the typical bit by bit configuration in a typical FPGA. This saves silicon area that can then be used for computation. A further benefit of this is that fewer bits need to be moved onto the chip to specify a new configuration, allowing faster reconfiguration times than with a typical FPGA. The Colt includes some aspects of DSPs by including a number of computational resources that are specialized for mathematical functions. A 16x16 fixed point multiplier has been incorporated in the design to enhance performance for mathematical operations. Further, the basic computational elements (Functional Units or FUs) include additional hardware, such as a barrel shifter, to aid in floating point processing as well as some integer multiplication operations. Colt architectural features and implementation will constitute a second major thrust of this work.

7

1.4 Contributions & Claims

There are two contributions to the Configurable Computing Machine (CCM) field that will be presented in this thesis. The first is the Wormhole Run-Time Reconfiguration paradigm. More than a simple extension of the traditional FPGA concept, Wormhole RTR explores new ground in the use of reconfigurable logic. Through the implementation of Wormhole RTR on the Colt CCM, improvements have been made with regard to microprocessor and/or FPGA performance in terms of:

- Execution Overhead

- Configuration I/O Rate

- Broadcast Programming

- Partial Run-Time Reconfiguration

- Degrees of Concurrency

- Global Routing Complexity

- Overall Speed of Reconfiguration

The second contribution is the design and development of the Colt CCM itself. There are many features that have been designed into this device that are unique. For one, it is the first device to incorporate Wormhole Run-Time Reconfiguration. All of the various resources available on Colt have been designed for use with the Wormhole RTR paradigm and so stand as examples of how this can be done. Also, Colt incorporates the idea of the Skip Bus. This segmented interconnection scheme has been superimposed over the normal mesh topology in

order to allow the greater connectivity required to map data flow graphs while not consuming vast amounts of area for routing. Finally, Colt is word-oriented rather than bit-oriented, and it has been designed with complex calculations in mind. No commercial CCM devices existing today can claim a word-oriented architecture. The benefits of this improvement alone with regard to contemporary FPGAs are seen in terms of:

- Computational Density

- Overall Speed of Execution

- Configuration I/O Rate

- Configuration Storage Density

- Routing Overhead

- Operating Clock Frequency

- Overall Speed of Reconfiguration

- Overall Speed of Execution

The Wormhole Run-Time Reconfiguration paradigm that is discussed at length in Chapter 3 has many promising characteristics that could be applied to a much broader class of computing platforms than simply traditional FPGAs. The characteristics listed here are quantified in Chapter 6; however, there are several more that are not so easily quantified, particularly as they might apply to designs larger than a single Colt device. These have been expounded upon in Chapter 3 and throughout this thesis as possible points of departure for future research. Likewise, the Skip Bus and some other Colt specific features will be difficult to

quantify until such time as the completed system has been built. It will remain for those who follow to judge the merits of these contributions.

1.5 Dissertation Outline

This work will describe a solution to the problem discussed in Chapter 1. Chapter 2 will begin with an overview of existing technologies, focusing mainly on commercial products. Some commentary on the relative merits of the various approaches will be included. This is followed by an overview of the data flow computing paradigm and some of the merits and drawbacks thereof. Chapter 3 introduces the particular implementation of the data flow paradigm used for this research. A discussion of the implementation strategy indicates how a limited data flow paradigm can be readily mapped to an FPGA style architecture giving benefits for communication overhead and pipelined implementation. Following this will be a discussion of the Wormhole Run-Time Reconfiguration (RTR) method. The method is expounded upon at length, with a great deal of time spent in delineating the benefits and how it fits in with existing research and technology. Chapter 4 presents the Colt architecture, a chip that has been designed based on the Wormhole RTR concept. An overall system perspective of the Colt is presented followed by sections on the major units of the chip with some features of their design that have been incorporated for various computational tasks. Chapter 5 discusses several key applications that illustrate various concepts of the Colt architecture. These include the implementation of a simple dot product of two vectors, and an example of floating point arithmetic. The chapter concludes with an example of conditional data flow execution. Chapter 6 gives a comparison of

Colt performance versus other architectures both in terms of raw computing power and of normalized measures of performance. Chapter 7 gives ideas for future research directions and design improvements that could be made in the Colt architecture and some ideas for the larger system. The first appendix contains an in-depth look at the design of the entire chip, discussing design tradeoffs, state machine implementation and critical path design of each unit on the Colt. The second appendix contains documentation and source code required for the implementation of a floating point multiplier on Colt.

2. Background

This chapter discusses various implementation approaches that could be taken to solve the DSP problem and their relative merits. Three possibilities are delineated: ASICs, DSPs and CCMs. The second half of the chapter begins with a discussion of recent trends in the design of microprocessors and the place of data flow computation in their evolution. Current data flow processing is reviewed as the chapter concludes.

2.1 Prior Work

2.1.1 ASIC Solutions

Assuming that the data paths, operators and pin functions are predefined, an Application Specific Integrated Circuit (ASIC) can be used to achieve the most efficient use of hardware resources in terms of speed, power, silicon area consumed or whatever other criteria are important for meeting the goals of a given task. Because of these factors an ASIC can also achieve the lowest cost per part for mass production usage. However, the design of an ASIC requires a large Non-Recurring Engineering (NRE) cost in terms of time and man power to produce final silicon. Further, once the design is completed, it is final. It is impractical to make changes in the finished silicon to correct errors in either the original concept or in the implementation of the original vision. Even in the absence of any design flaws, an ASIC is

limiting in that it allows no variations in functionality beyond those foreseen at the beginning of the design process. If better algorithms or techniques are discovered, they cannot be applied in the existing parts.

This final point is really the best case against using an ASIC for prototyping any system. There may be different algorithms that the end users of the system may wish to employ. Indeed, the exact algorithms that will be needed may remain unknown until a prototype is built. Such is the case for the communications system that this platform is targeted for. Furthermore, as will be explained in Section 2.1.3, though one of the advantages of an ASIC implementation is the ability to tailor it to a specific application, the fixed nature of an ASIC can ultimately lead to poorly utilized silicon area even for the application to which has been tailored. As points of reference, two specific ASIC devices will be examined in the remainder of this section.

2.1.1.1 Harris Semiconductor HSP45256

It is relatively common to find DSP kernels implmented as ASICs. One such chip is the Harris Semiconductor HSP45256, which is being used for prototype systems at Virginia Tech [4]. This chip can be used to implement FIR filters of the type needed in power of two sizes including 32, 64, 128 or 256 taps depending on the bit width of the input data. Input data widths of 1, 2, 4 or 8 bits are supported. To determine the number of taps that can be implemented, divide 256 by the input data bit width. For purposes of an FIR filter as described in Section 6.1, the spreading code is programmed as a single bit "reference" for each tap in the filter.

---

4   Harris Semiconductor, *Harris Semiconductor Digital Signal Processing Databook*, 1994, pp. 7-21 - 7-33.

The Virginia Tech Glomo project [5] requires at least 12 bit samples and a 60 tap FIR filter with 7.86 million samples per second arriving from the antenna. Using creative reconfiguration of the ASIC, four different configurations could be used to perform the desired computation which would allow approximately 10 million samples per second to be processed. Of course, additional memory and control hardware would be required.

2.1.1.2 Logic Devices Incorporated LF2242

The LF2242 is an FIR filter designed to implement a low pass filter [6]. The weights of the filter are fixed so that it implements low pass filtering exclusively. This chip, in contrast with the Harris chip, demonstrates the tradeoff of performance verses flexibility. Both chips operate at 40 MHz; however, the LF2242 accepts 12 bit samples into a 55 tap FIR as opposed to 8 bit samples into a 32 tap FIR in the Harris chip. In the Harris, the weights are programmable, but in the LF2242 they are fixed. Greater flexibility comes at the expense of overall performance. The LF2242 is not suited for some tasks in the Virginia Tech Glomo project, such as the remote unit FIR application, since the spreading code vector could not be programmed into it.

2.1.2 DSP Solutions

Digital Signal Processing (DSP) chips have been developed in recent years in an attempt to address the large computational requirements of digitally based communications systems and

5 Rappaport, T. S., Athanas, P. M., Reed, J. H. and Woerner, B. D., "A High Capacity Adaptive Wireless Receiver Implemented with a Reconfigurable Computer Architecture," *ARPA Principle Investigators Meeting*, Fort Lauderdale, Florida, July 1995.
6 Logic Devices Incorporated, *DSP Data Book*, July, 1995, pp. 2-3 - 2-10.

some other types of processing. These chips are modeled after conventional microprocessors with extra silicon devoted to specific operations in order to achieve higher performance. Typically, fixed or floating point computations can be handled with equal ease and in comparable amounts of time. A multiply-accumulate (MAC) unit is usually included for the computation of dot product like formulations.

The main argument against using a DSP as the processing element of a computationally intensive system is that DSP processors, and general purpose processors for that matter, focus on sequential execution of applications. In so doing, DSPs devote most of their silicon area to non-computational tasks. Microprocessor's of today incorporate cache, memory I/O units, instruction decoders, pipeline and superscalar control units, large register files, etc.; all in an effort to bring the operands to the arithmetic units faster so that the program will execute faster. An effort has been made by Flynn to empirically quantify the distribution of the amounts of silicon used on a general purpose processor for all the various functions that people have come to expect [7].

Though Flynn uses the relative sizes to describe a floor planning methodology, the same information can be used to show the amount of overhead that is incorporated into today's processors. The dimensions for various units that are described are given in terms of a unit A, defined by Flynn as being 4 million λ^2. The unit λ is used in VLSI terminology to abstract the minimum length of a transistor in a given process. This unit was proposed by Mead and Conway as a means of dealing with the problem of quantifying the size of a circuit in the face of ever

---

7   Flynn, Michael J., *Computer Architecture: Pipelined and Parallel Processor Design*, Boston, MA, Jones and Bartlett Publishers, Inc., 1995, pp. 90-99.

changing process specifications [8]. It is defined as the distance by which a geometric feature on any one layer of implementation may stray from another geometric feature. For example, in a 0.8μm process the minimum length of a transistor is 0.8μm, which is defined to be 2λ, so for that process λ = 0.4μm. There is some variation in the relative distances between features from process to process, but on the whole this system gives a good means for comparing relative circuit sizes between two different processes.

Flynn defines a basic RISC style microprocessor with 32 registers of 32 bits each, an integer ALU and an integrated floating point unit. It is assumed that the processor has an instruction pipeline of 4 or 5 stages to implement a load/store architecture that are referred to as being typical of the early 1990s. He states that the complexity of the instruction set will only affect the relative area of the control units modestly. Below are the relative sizes of various units as he tabulates them.

---

8   Mead, C. and Conway, L., *Introduction to VLSI Systems*, Series in Computer Science, Reading, MA, Addison-Wesley, 1980.

Table 2.1 - General Purpose Processor Unit Sizes [7].

Integer Unit	
Unit	Area
Integer ALU (32 bit)	1.0A
Bypass	0.15A
Integer Registers	1.0A
Shifter	0.5A
Incrementor	0.4A
I-fetch/PC unit	0.85A
2 Translation Look Aside Buffers (For Memory Paging)	2 * 3A
Decode and Control	1.0A
Cache Controller	1.0A
Bus Logic	2.0A
Store Buffer and Bypass	1.0A
Load/Store Byte Support	0.2A
Clock Generator	1.0A
Subtotal	16.1A
Floating Point Unit	
Register File	1.0A
Adder	13.5A
Multiplier	20.3A
Division Support	3.0A
Subtotal	37.8A
Complete Processor	
Integer Unit	16.1A
Floating Point Unit	37.8A
Latches and Buses	27A
Total Area	80.9A

Table 2.1 indicates that in the integer processor the actual data processing units, consisting of the ALU, shifter and incrementer occupy 1.9A of the silicon resources. The one unit that is conspicuously absent from the table is cache. Flynn computes an area estimate of

17

15.9A for a rather modest 4K cache. For the processor as a whole, adding in the active units in the floating point hardware, $(1.9A+36.8A)/(80.9A+15.9A) = 40\%$ of the processor is actually used for computation of data. The rest of the area is used to feed that hardware at the fastest rate possible with new data. In effect, it is overhead needed by the control driven Von Neumann architecture to keep the data computation rates at acceptable levels.

For the purposes of many DSP applications it may not be necessary to implement floating point arithmetic at all, in which case the amount of silicon dedicated to computation is only $1.9A/(80.9A+15.9A) = 1.96\%$. This is an abysmally low number for silicon utilized. To be fair, if it is known that floating point computation will not be required, a processor that does not have a dedicated floating point unit could be chosen. Using Flynn's approximation of adding 10% to the unit area for latches and another 40% for buses and control, we get an approximate total area estimate of $16.1A*150\% = 24.15A$ for a purely fixed point processor. Thus, the amount of silicon devoted to computation in such a processor would be $1.9A/(24.15A+15.9A) = 4.7\%$. Note that even these numbers are generous because 4K is a rather conservatively sized cache. Also, this is just the level 1 cache, which is usually augmented by a level 2 cache of 256K or more. Again, the problem is obvious: to much silicon is being used for overhead processing and not enough for computation. This is the main argument against general purpose and DSP processors.

18

2.1.2.1 Analog Devices ADSP-21060 SHARC

The 21060 Super Harvard Architecture Computer (SHARC) typifies a contemporary high-performance DSP [9]. Running at 40 MHz, the chip achieves 120 MFLOPs peak with a sustained throughput of 80 MFLOPs. IEEE 32-bit floating point format is supported throughout the system. A multiply and add of two independent sets of floating point operands can be issued and completed in a single clock cycle, achieving 80 MFLOPS operation. For the purposes of computing Fast Fourier Transforms (FFTs), an independent multiplication can be done at the same time as the addition and subtraction of the same set of operands; all in one clock cycle giving 120 MFLOP peak performance. Data is supplied to the computing engine by a Super Harvard Architecture using three buses. One bus (DM) fetches data from memory each clock cycle. The second bus (PM) can fetch either a data operand from data memory or an operation from program memory if there is a cache miss. The third bus fetches op codes from the cache when possible. 4 Mbits of RAM are built into the chip, which can be used in several different word widths including 16, 32 and 48 bits. Using the 32-bit configuration, there are 128K words of onboard RAM. There are also extensive communications and I/O resources; including independent DMA controllers, serial ports and additional parallel data ports.

This chip was chosen by the communications groups in the Virginia Tech Glomo project for use in implementing their prototype systems. By making a number of simplifying assumptions in the algorithms, they have been able to produce a system that can operate at a maximum data rate of 30Kbps; less than ¼ of the 128Kbps rate required in the final system. It is

---

9   Analog Devices, *DSP/MSP Products Reference Manual*, 1995, pp. 2-215 - 2-250.

difficult to determine the amount of silicon resources that have been used in the 21060 to implement the floating point unit(s); however, accomplishing a full floating point multiplication or addition in 25 ns is quite a feat. The 21060 can do both, indicating the amount of resources that have been thrown at the problem. In the 60 tap FIR filter example needed for the base station, the SHARC can do a MAC operation every clock cycle, so it could do 40MHz/60 = 0.67 million evaluations per second; a bit short of the required 7.86 million evaluations per second. To be fair, the SHARC can do this using full floating point weights and data points, whereas the other benchmarks presented here can only operate with integer operands and in some cases the weights may only be one bit wide.

2.1.3 FPGA Solutions

An interesting alternative to ASIC and DSP devices are Field Programmable Gate Arrays (FPGAs). These chips are programmable at the logic gate level. In a standard microprocessor, the logic gates which comprise the functionality are set at the time of manufacture. Groups of gates perform dedicated functions and the functionality of these gates never changes over the life of the chip. Instead, the microprocessor iterates through the Von Neumann fetch-execute cycle. Instructions are read (fetched) from memory by the microprocessor in an order specified by the programmer. These instructions dictate which set of gates to apply to the data at the current time step.

In contrast, the gates in an FPGA are not connected in a particular configuration at the time of manufacture. Instead, the chip can be loaded with new configurations at will. This allows the operations performed on the data by a given set of gates to be tailored to a particular

application. The exact function required can be performed with the exact precision required. A microprocessor typically performs functions on a fixed bit width (8, 16, 32 or 64 bits), regardless of the number of bits of precision required. The Virginia Tech Glomo project, for example, requires 12-bit fixed point data operands. The mathematical operations performed on these can be guaranteed never to exceed certain bounds. If a set of 8 such numbers are added, the precision requirements for the sum can be guaranteed not to exceed 15 bits. Knowing this, the FPGA programmer can create mathematical units on the chip that operate with 15 bit sums rather than some number of bits set by a power of two. The end result is a net savings in the number of gates required to implement the algorithm and hence the final cost in terms of silicon consumed.

However, the savings in silicon is not quite as dramatic as has been implied. A gate in an FPGA consumes more area than the same gate in an ASIC or microprocessor. There are two factors contributing to this: programmability and routing. First, in order to provide the ability to program the gates with arbitrary functions it is necessary to include memory storage elements that set the function that a gate will perform. These storage elements are not required in an ASIC since the function can be fixed at design time. Alternatively, there are FPGAs in which a single functionally complete gate is used throughout the chip (perhaps they are all NANDs), thus no memory elements are required to set the function. However, more gates are then required to perform arbitrary functions. These FPGAs also suffer even more greatly from the second problem of routing.

The flexibility of connecting gates in programmer selectable configurations comes at the expense of added silicon for routing resources. Extra wiring must be added to allow sufficient flexibility in gate connections. Further, more memory elements must be added to the chip to

21

control the connection of these extra wires. The memory elements control decoders, multiplexers or tri-state buffers to steer the bits of the computation from one gate to the next. Obviously these are not concerns in an ASIC or microprocessor since the designer knows exactly where the data bits must go at design time. Multiplexers and the like are used in fixed function chips; however, and that addresses another area where FPGAs can prove to be the more efficient of the two to use.
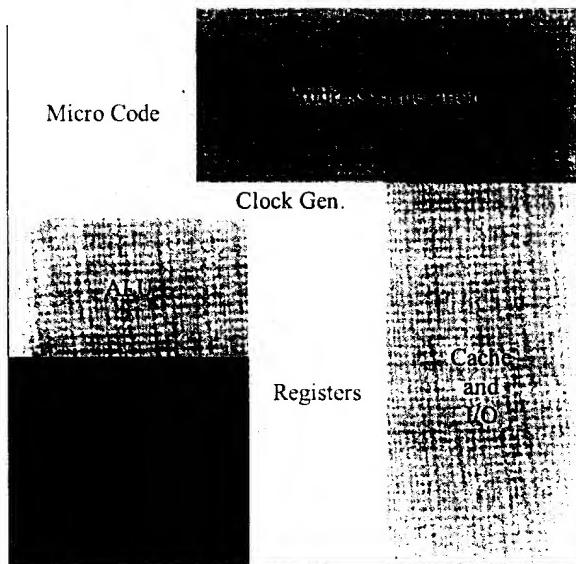
Figure 2.1 - Microprocessor Floorplan.

Figure 2.1 shows an example of the floorplan for a microprocessor. Different areas of the chip are devoted to different functions. Each function of the processor is allocated an area of silicon and the designer optimizes a unit to fit into that space. This partitioning of silicon area is hierarchical and extends down into each unit with finer and finer degrees of detail. As discussed above, the functions that can be performed on the data stream in an ASIC or microprocessor are fixed at design time. The instructions direct the processor to perform a given function at a given

22

time step and the operands are forwarded through a series of multiplexers to the hardware performing the function. In a processor with a single thread of execution (not superscalar), the rest of the execution hardware remains idle until an instruction is executed that requires it.
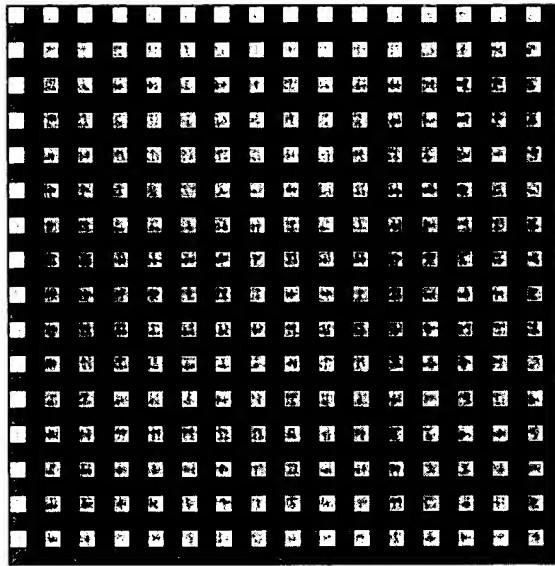
Figure 2.2 - FPGA Floorplan.

A sample floorplan for an FPGA is shown in Figure 2.2. The area of an FPGA is devoted to an array of identical cells. These cells typically perform one or two functions that can be any function of 2 to 4 bits using a programmable look-up table. The cells also generally contain one or two usable flip-flops. Nearest-neighbor connections between cells are always supported, and there are a variety of other routing resources available for connections to more distant cells as well as special wires that can be used for clock distribution and system reset. The key to the idea is that the FPGA can be reconfigured to perform different functions at different times. Whereas the silicon resources on a standard microprocessor are fixed at design time and then the

appropriate function is selected at run time, an FPGA can be reconfigured on the fly to perform whatever function is necessary.

Thus, though an FPGA requires more silicon area per gate to implement a given function, that function can be changed over time. There are two applications of this for the designer. The first is the traditional role of an FPGA as a platform for rapid prototyping of new designs. The design cycle of an ASIC demands long periods (up to several weeks) of latency between submission of the design to the fabrication facility and the return of a finished part. The wait slows the design cycle, and more significantly, the fabrication process is very costly. An FPGA allows a designer to quickly try many different designs in a single day, using the same part and merely downloading different configurations into the chip. Further, changes to the design are essentially free since they only require a recompilation of the design into the binary format required for the FPGA. Rapid prototyping has become one of the main applications for FPGAs today.

An emerging application of FPGAs is Run-Time Reconfiguration (RTR). An FPGA can be used to implement *temporal* multiplexing of functionality using the same hardware resources in addition to the *spatial* multiplexing of resources to which a microprocessor designer is restricted. In effect, a sort of hardware page swapping of configuration data can be performed at run-time. By tailoring the allocation and utilization of hardware resources to the particular problem or section of an algorithm that is currently executing, the FPGA programmer can achieve performance speedups over generic (microprocessor style) execution units. Swapping different hardware configurations into an FPGA at different times in the execution of the algorithm can allow this customization to produce a marked increase in processing efficiency.

RTR in FPGAs is a field of intense research interest at present. The application of RTR to the pipelined processing of data flow graphs is a large part of the thrust of this work.

One final advantage of an FPGA implementation of the system required for prototyping work is algorithmic design flexibility. While it is undeniable that a microprocessor or DSP can be programmed to perform any algorithm of interest, the speed at which it will execute can be a limiting factor. An ASIC can achieve the fastest possible execution speed, but the algorithm used and the method of implementation are fixed at design time. An FPGA allows near ASIC execution speeds while still allowing the flexibility of experimentation with different implementation strategies.

The remainder of this section provides an in-depth description of several contemporary FPGAs in an effort to establish a reference point for the architectural styles that are currently being employed. In addition, since Chapter 6 will be using some of this information to generate a number of figures for comparison, these will require a detailed understanding of the architectures involved. The Xilinx XC6200 receives particular attention due to its unique configuration style.

2.1.3.1 Altera FLEX 8000

The Altera FLEX 8000 is an example of a typical FPGA [10]. It consists of an array of cells, which are called Logic Elements (LE). Each LE contains a programmable look up table that can generate any function of 4 variables in addition to a flip flop. The LEs are grouped into Logic Array Blocks (LABs) of 8 LEs each. Special carry logic connects the LEs in a LAB and all

the LABs on a given row of the array so that adders can be implemented efficiently in terms of routing and without undue carry propagation times. Altera's largest 8000 series device is the EPF81500 which contains 1296 LEs. It has 6 rows of 27 LABs each. Each LE feeds a wire in the corresponding row that spans the entire width of the device. Thus, in the EPF81500 there are (27 LABs) * (8 LEs/LAB) = 216 wires spanning the width of the device. Between the columns of a row there are 16 wires that span the entire height of the chip. Each of these vertical connections can be driven by one of several LEs that is connected to it via a tri-state buffer.

In the Altera devices, an obvious design choice has been made in favor of creating wires that span the entire chip rather than creating a segmented wiring scheme or one in which the wires have limited length. In a segmented scheme switches are required to route a signal from one wire segment to the next. The switches add to the propagation delay of the circuit and slow down the maximum operating rate. Limited length wires obviously limit the distance and number of cells over which another signal can be sent. Altera chose to use a greater amount of silicon for routing to reduce propagation delays while not limiting connectivity.

The FLEX 8000 series chips support six different configuration modes involving a number of different independent and controlled programming strategies using either a parallel or serial interface. The entire chip must be reconfigured each time it is programmed, and the speed of reconfiguration is listed as less than 100 ms.

---

10 Altera Corporation, *Altera 1995 Data Book*, 1995, pp. 37-93.

2.1.3.2 Xilinx XC4000

The Xilinx XC4000 series of chips has become a mainstay of FPGA development [11]. The basic cells used are called Combinational Logic Blocks (CLBs) and they are relatively complex compared to some others. Each CLB can generate two functions, F' and G', of 4 variables each, along with a third function H' that is a function of F', G' and a third independent variable. Also included in the CLB are two flip flops that can be used for storage. As with the Altera device, a fast carry logic chain runs through the CLB for implementing wide adders, decrementers, etc., with shortened propagation delays. An interesting difference is that the look up tables used to generate the logic functions can be used as a RAM of either 16x2 or 32x1 bits.

Table 2.2 - Xilinx XC4000 Series Specifications.

Xilinx Part	XC4005	XC4010	XC4013	XC4025
Appr. Gate Count	5,000	10,000	13,000	25,000
Number Of CLBs	616	400	576	1,024
Number Of IOBs	112	160	192	256
CLB Array Dimensions	14x14	20x20	24x24	32x32
Number Of Bits Of Configuration Data	81,372	178,136	247,960	422,168

The CLBs are laid out in a square array of 32x32 for a total of 1,024 CLBs in the XC4025; Xilinx's largest device in this family. There are several different types of routing resources used to connect the CLBs. Single-length lines jump from one row-column intersection to the next, double-length lines make a connection between every other intersection, and long lines go across the entire height or width of the chip in either the vertical or horizontal direction.

---

11 Xilinx Corporation, *The Programmable Logic Data Book*, 1994, pp. 2-4 - 2-43.

Single and long lines can connect at the row-column intersections, but double length lines do not connect to the other routing resources at all.

Each CLB in the XC4000 family requires about 430 bits of configuration data per CLB and its interconnect. There are six modes of programming these chips including 3 master modes in which the chip configures itself, 2 passive modes in which a controller programs it and a slave mode in which multiple chips can be daisy chained together. For the XC4025 device there are a total of 422,168 configuration bits that must be stored in EPROM and then loaded into the chip. In master serial mode, a clock of up to 10 MHz loads the data onto the device, thus requiring $422,168/10\text{MHz} = 42.2$ ms to reconfigure the device. The master parallel mode does not configure the device any faster and merely loads 8 bits of data from 8 dedicated input pins and then serializes them at the same rate.

2.1.3.3 Altera EPF8050M

The Altera EPF8050M is the first FPGA type device discussed here that supports partial run-time reconfiguration [12]. This is actually a Multi-Chip Module (MCM) of four EPF81188 die. The EPF81188 is one of the Altera FLEX 8000 family of FPGAs containing 1008 LEs arranged in 6 rows of 21 LABs each. 160 of the 180 die pads on each of the EPF81188s are connected through a full crossbar (called a Field Programmable Interconnect (FPIC) device) to the other EPF81188s as well as 360 I/O pins on the MCM package.

The EPF8050M is interesting because of the partial reconfiguration capabilities. In "passive" programming mode a host computer can be used to configure the individual FPGAs in

28

the MCM either in parallel or independently. This allows a somewhat coarse grained approach to run-time reconfiguration. While each EPF81188 must be configured as a unit, the fact that they can be configured independently means that ¾ of the MCM can be processing data while the other ¼ of the hardware resources are being reconfigured to perform the next step in the task, or an entirely an unrelated task. Hiding configuration latency time in this way effectively pipelines the act of reconfiguring the FPGA and the active use of a configuration; keeping the data moving through the system as quickly as possible.

2.1.3.4 Xilinx XC6200

The XC6200 series chips are the latest additions to the Xilinx line [13] as of this writing. They are based on a hierarchical mesh architecture whose basic element of computation is called a cell. A cell contains a flip flop and can compute an arbitrary function of two Boolean variables. These cells are laid out in a square grid pattern on the chip. The cells are connected by what might be termed a logarithmically structured mesh network. Some specifications for this chip line are shown in Table 2.3. Note that at this time the XC6216 is only part nearing market readiness and the others are planned products.

12 Altera Corporation, *Altera 1995 Data Book*, 1995, pp. 95-115.
13 Stansfield, Anthony and Page, Ian, "The Design of a New FPGA Architecture," *FPL95*, Lecture Notes in Computer Science, 1995.

Table 2.3 - Xilinx XC6200 Series Specifications [13].

Xilinx Part	XC6209	XC6216	XC6236	XC6260
Appr. Gate Count	9k-13k	16k-24k	36k-55k	64k-100k
Number Of Cells	2304	4096	9216	16384
Number Of IOBs	192	256	384	512
Cell Array Dimensions	48x48	64x64	96x96	128x128

Each cell has connections to and from the four local neighbors. The cells are grouped in 4x4 squares of 16 cells each. Each of these 4x4 blocks of cells forms part of a larger 16x16 block of cells that is comprised of a 4x4 array of the 4x4 cell blocks. The 16x16 block of cells is then laid out in a 4x4 array to form a 64x64 block of cells. This is where it ends for the XC6216 chip, but the pattern scales up for the larger chips.

The cells are grouped into blocks in this way so that logical and physical boundaries can be made for the "fly overs." A fly over is a wire that runs over top of the cells that can be used for routing without using any cell resources. The fly overs are sized according to, and are associated with, each logical cell block mentioned above. A four cell long fly over wire runs over top of each row and column of every 4x4 cell block. The fly over provides extra connectivity between cells. A fly over 16 cells long runs over the entire length of every row and column of a 16x16 cell block. Likewise, each row and column of a 64x64 cell block has a fly over running its entire length. So, any given cell on a XC6216 chip has length 4, 16 and 64 fly overs running over them going north to south and east to west. The naming convention for these connections are labeled N4, N16, N64 and so on for north going fly overs of length 4, 16 and 64 respectively. The fly over connections greatly enhance the connectivity possibilities on the chip;

however, an arbitrary cell cannot directly access all of the fly overs that are running over top of it.

To describe the restrictions a better understanding of the standard cell is needed.
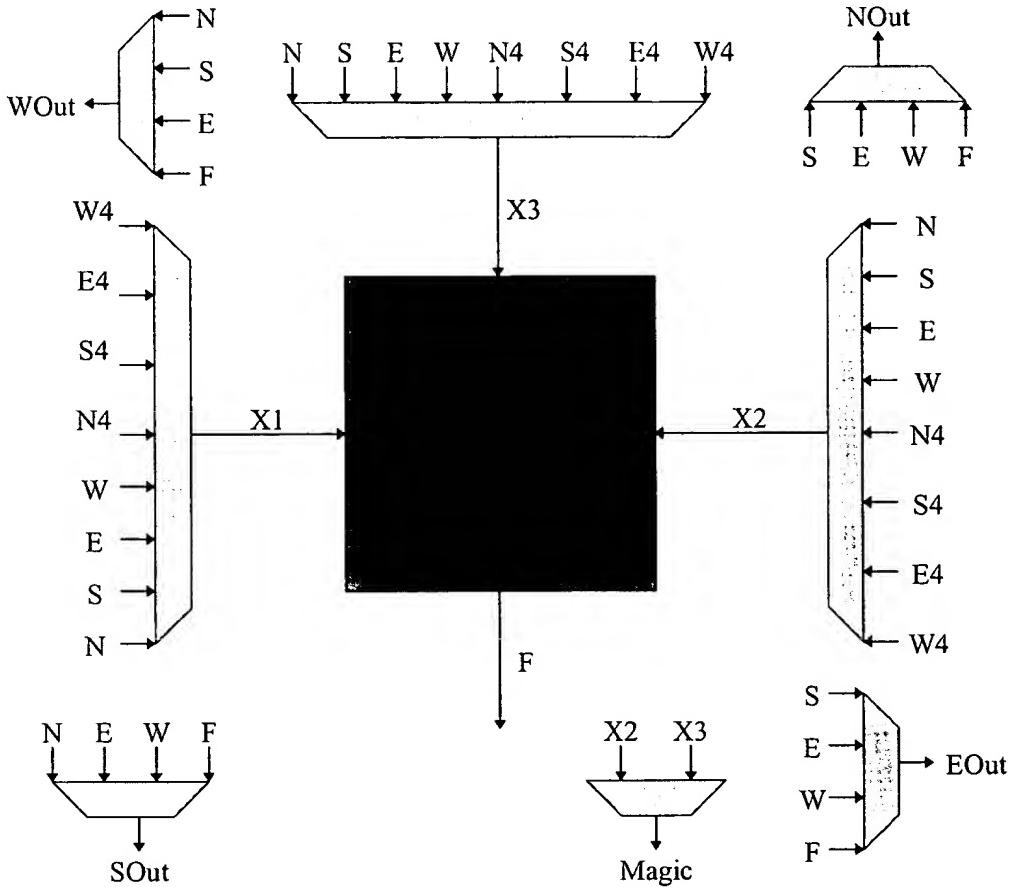
Figure 2.3 - XC6200 Standard Cell Connectivity.

The connectivity for a basic cell is shown in Figure 2.3. The neighboring signals, denoted

by N, S, E and W can be selected to drive the function unit and they can also be routed through

the cell without stopping at the function unit. The N4, S4, E4 and W4 signals are the first level

of length 4 fly over wires discussed above. Note that the basic cell can only get inputs directly

from the local connections and from the length 4 fly overs. The Function Unit output goes out on

local connections only. The fly over connections can be driven only from logical block boundaries. All 4x4 cell block boundaries have special multiplexers to allow the cells bordering the boundary to drive the length 4 fly over for that row or column of the block. These boundary multiplexers can also allow the length 16 or length 64 fly overs for that row or column to drive the length 4 fly over. Also, note that each cell has a Magic output. This output can only be used in certain functional configurations of the cell. If the Magic output can be used it provides direct connections to two of the 4x4 cell boundaries so that the cell output can be sent on to the length 4 fly overs or other destinations. Note that the Magic output cannot carry the Functional Unit output signal.

At higher order block boundaries, such as for a 16x16 cell block, there are additional multiplexers to allow cells and fly overs bordering the logical boundary to drive the length 16 fly over for a given column or row. Likewise, similar multiplexers are present at 64x64 cell block boundaries for chips larger than the XC6216.
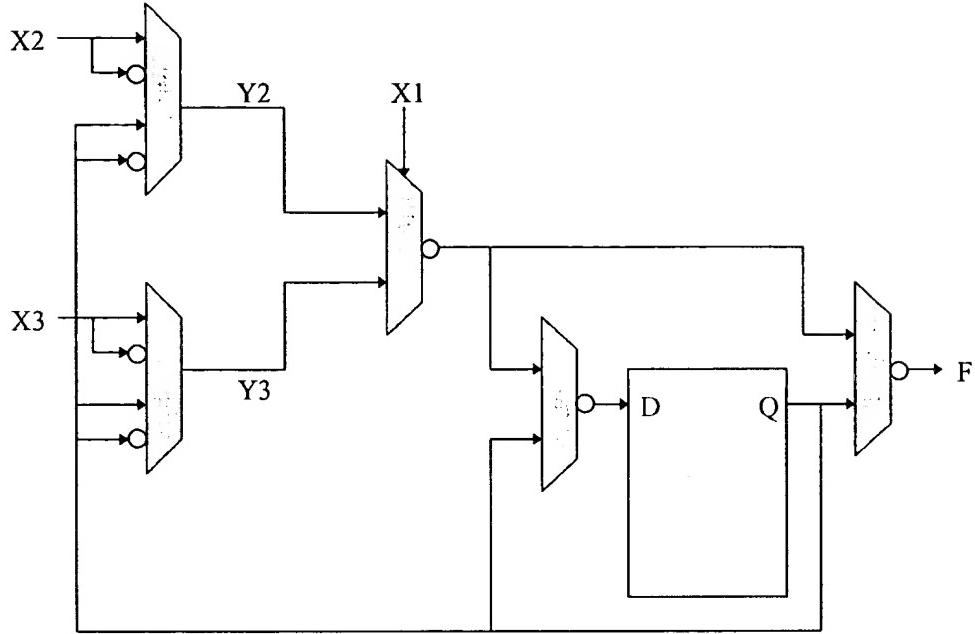
Figure 2.4 - XC6200 Function Unit.

As in the Xilinx XC4000 series devices, IOBs are used to connect the silicon itself to the

packaging pins. IOBs are arranged around the entire circumference of the array of cells. Hence,

in the XC6216 part there are 4*64 = 256 IOBs on the chip. However, all of these IOBs aren't

necessarily connected to pins. The number of IOBs that are connected to pins depends on the

chip packaging used. The current options allow for 84, 240 and 299 pin configurations. In cases

where the number of IOBs is greater than the number of pins, some IOBs are not connected to the

outside and can be used to drive control signals for IOBs that are connected to pads.

The XC6200 can be configured in active and passive modes through parallel or serial

input ports as with the other chips, but the most interesting method available is random access.

Through the 32-bit data port the configuration bits of each cell in the chip can be read or written

just as is done in a normal RAM. This gives the ability to reconfigure areas of the chip with a

much finer degree of control than is available in the EPF8050M, which must be configured in ¼ total resource increments. Further, large areas of the Xilinx chip can be programmed with identical programming information through the use of a wildcard register that allows the user to make some of the address bits for the configuration write behave as don't cares. For example, if the wild card register has the value 0100 and the user writes to address 0001, then on chip addresses 0101 and 0001 are both written to simultaneously. This is of great advantage for large repetitive structures such as multipliers. It is important to note that at most 5 bits can be used as wild cards in this way, so that at most 32 columns of a XC6200 series device can be programmed simultaneously with the same data. Also, the wild card mask applies only to decoding the column address for the cells to be written. The row address within the column uses broadcasting with a finer detail of control through the use of a second mask register. This second 32-bit mask register individually selects which cells in the column will be programmed using a one hot assignment of bits in the register to cells in the column.

Calculating the time needed to reconfigure a XC6200 series device is difficult since the exact time depends on the extent to which the broadcast programming features can be used. A range can be established, however. On the fast end of the range, up to 32 columns can be programmed simultaneously, with all cells in that column receiving identical programming information. The minimum write cycle time is rated at 40 ns, and each cell requires 3 bytes of programming information, that presumably can be configured using a single 32-bit write. So, in a XC6216, which has 64 columns, it would take one cycle to write to the column mask register, another to write the row mask register and a 32-bit write to set the configuration for the first 32 columns. Then an additional write to the column mask register followed by a final 32 bit write of

34

configuration data to set the last 32 columns. This is a total of 5 write cycles of 40ns each, establishing a lower bound on configuration time of 200ns to program the entire chip to do exactly the same thing. On the slow end of the scale, the case of every cell having a unique configuration can be considered. For this case, each cell would need to be written to individually, thus 4,096 cells each requiring one 40ns write would take 122.88μs. So, a fairly wide range of possible programming times of 200ns to 122.88μs has been established.

A final feature of a chip in the XC6200 line is the ability to internally drive control signals allowing the designer to set up a chip so that it can program itself. It is possible for part of the chip to be configured with a sort of boot strap controller that can dynamically reconfigure other parts of the chip to perform different tasks. Reconfiguration data must still come through either the serial programming interface or through the 32 bit parallel interface and so the rate of reconfiguration is still limited by the bandwidth of one or the other of these data paths. None the less, this is an intriguing capability. Not only can the hardware "page swapping" exhibited in the Altera EPF8050M be used, but it can be performed by a state machine of the designer's choice. This allows the designer to configure the chip to perform a fetch-execute cycle with configuration information, just as is done in a normal microprocessor with instructions, but on a much grander scale.

2.1.4 Hybrid Solutions

In some sense, the Colt CCM could be called a hybrid solution to the DSP problem. In this section, other hybrid solutions are described in an effort to show some of the range of ideas that have been expended on this problem. More importantly, the variety of approaches should

35

help illustrate that Colt is more of an extension of the FPGA field (a CCM) than an eccentric approach to the problem.

2.1.4.1 Infinite Technology Corporation RAD5A4

The RAD chip is a proposed 100 MHz single chip parallel control flow machine [14]. As shown in Figure 2.5, the RAD chip consists of 4 Macrosequencers and an Arbiter module. The Macrosequencers, as they are called, are small processors each having associated code storage, dual register sets and arithmetic units.
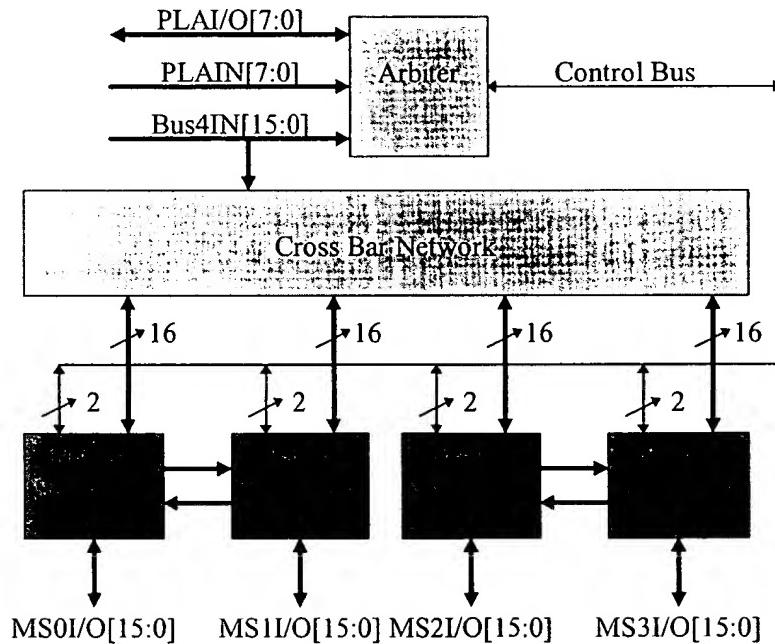
Figure 2.5 - RAD5A4 Chip Architecture.

---

14 Infinite Technology Corporation, *RAD5A4 Preliminary Data Sheet.* April 1995.

Each Macrosequencer has eight possible sources for 16-bit input data. Five of these come from the cross bar network, which gives access to the output of any of the Macrosequencers as well as data from an off-chip bus (Bus4IN[15:0]). Another input source comes from a dedicated I/O port that is uniquely associated with each Macrosequencer and gives it direct access to off-chip resources. These are labeled MSxI/O[15:0]. The seventh input source comes from the Macrosequencer's mated pair and is shown by the complementary connections between Macrosequencers 0 & 1, and 2 & 3. This last set of connections allows these two pairs to be easily linked together to perform 32-bit operations. Finally, a Macrosequencer can store a constant and use that as a data source for calculations.

The Macrosequencer internal architecture consists of a controller and an arithmetic unit. The controller allows one or two small programs to be loaded at power-up and to be reconfigured later. The Arbiter can then selectively trigger these to start running using the Control Bus. A total of 32 instructions can be programmed into a controller, either of the two programs can then either run once, run indefinitely, be stopped by the Arbiter or be stopped by a condition within the Macrosequencer. The instruction set directly supports only integer operations, and includes 16x8 and 16x16 multiplication, 32 and 16-bit addition, 16x8 and 16x16 multiply accumulate, shifting and all the logical operations of two bits such as AND and OR, as well as branching and conditional instructions.

The arithmetic part of a Macrosequencer consists of an input selector, output selector, and units exactly corresponding to the instruction set mentioned. The Multiply Accumulate (MAC) unit is a 16x8 multiplier that can produce the 32-bit result of a new 16x16 multiplication every three clock cycles. If MAC operations are being performed, then $2n+2$ clock cycles are required

37

for n 16x16 bit products. Also part of the arithmetic unit is a set of 32 addressable registers, each 16-bits wide. The registers can be accessed directly, or an automatic increment or decrement mode can be used to access them in sequence. These differ from the included 16 word x 16 bit data memory in that the data memory has dual read ports and a single write port, and it has more sophisticated indexing capabilities. The data memory has separate read and write indexes that can automatically and independently access a location and then jump by a specified offset to a new location. This is said to have applications in performing FFTs and DCTs where symmetric weights are used in the calculations.

The Arbiter is used to coordinate the operation of the four Macrosequencers and to communicate with off-chip devices. It is designed to function as a large state machine and consists mainly of two 32 input PLAs, some state registers and 16 output signals. Some of the inputs come from the PLAIN[7:0], Bus4IN[15:0] and the returning Macrosequencer control signals. Of the 16 outputs produced, 8 are used to control the Macrosequencers via the 2 bit Control Bus going to each. The other 8 output signals connect to the off-chip PLAI/O[7:0] bus shown at the top.

This chip is a standard MIMD machine under the Flynn taxonomy using a restricted mesh architecture for communications. The preliminary data sheet mentions that it can be reconfigured at run-time to execute different programs, but neither the time involved nor the methods of completing this process are given. It is however, an interesting, if somewhat obvious, utilization of the increasing silicon resources available in VLSI processes. If four simultaneous FIR filter calculations, as needed by the base station, were performed on this chip a processing rate of $4*100MHz/(2*60+2) = 3.279$ million evaluations per second could be achieved.

38

2.1.4.2 CMU Programmable Systolic Chip

As described by Fisher, et al. [15], the Programmable Systolic Chip (PSC) is a 25,000 transistor microprogrammable chip that is intended for use as a building block for large systolic array systems. It consists of three input ports and three output ports, each 8 bits wide. There are also three input and three output single bit ports used for control signals between neighboring chips. Inside the PSC is an 8-bit ALU and a Multiply Accumulate unit that can accept 8-bit operands and produces a 16-bit sum. There are sixty four 9-bit registers for data storage and microcode for the chip is stored in a sixty four word by 60-bit control store. Thus, the chip could be compared to an Intel 8057 with a MAC added, but the MAC makes the chip much more suited for use as a systolic array element. A system was constructed from nine of these chips and was used to implement both 1-D and 2-D convolution.

This was the first chip of its scale done at Carnegie-Mellon University and as such was an accomplishment in its own right. Fisher, et al., describes many lessons learned concerning design practices and design methodologies through the production of their chip. A lack of rigorous voltage, temperature and clock variation simulations caused significant electrical problems. Also, high level floor planning is recommended from the start along with Layout Verses Schematic (LVS) tools, to which, at the time, they did not have access. They suggest that features should be traded for die size to in order to increase yield and "fancy" dynamic RAM designs should be left to the professionals. Finally, they recommend that a full plan be created

---

15 Fisher, Allan L., Kung, H. T. and Sarocky, Kenneth, "Experience with the CMU Programmable Systolic Chip," *Microarchitecture of VLSI Computers*, Boston, MA, Martinus Nijhoff Publishers, pp. 209-222, 1985.

up front for simulation, documentation, testing and demonstration. All of these suggestions seem to be common sense, but in a university setting with scarce resources the temptation is to cut corners in favor of shorter turn around time.

Though the intended use of this chip is similar to that of the Colt, there are significant architectural differences between the two. First, the PSC was built around a stack based control driven architecture, whereas the Colt is built for use in a data flow environment. Secondly, the PSC uses no notion of a stream and even if it did, there is only one ALU and one MAC. Though it may be possible to use the two in parallel, it would be difficult to utilize different sections of the PSC for different computational tasks, or to switch the task of one section while the other remains running. These types of operations are handled effortlessly by the Colt chip and should be prerequisites for a reconfigurable computing candidate.

2.1.4.3 USC Multiprocessor DSP Chip

The USC Multiprocessor DSP chip resembles the Colt in that it consists of a mesh of processing elements intended to accelerate DSP type operations, attempting to achieve the highest possible computational power per unit area [16]. The chip can be configured into a 1 dimensional ring of PEs or a 2 dimensional mesh. Each PE is connected to the four nearest neighbors and receives instructions that have been broadcast from a global controller. Hence, it is an SIMD machine, and in that sense differs from Colt.

---

16 Chang, Chia-Fen, Sheu, Bing J. and Okada, Hiroto, "Design of a Multiprocessor DSP Chip for Flexible Information Processing," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. v-637 - v-640, 1992.

Each PE uses 16-bit buses for communication and contains a 16-bit Wallice multiplier producing 32-bit results as well as a 40-bit adder. Each PE occupies 4.1 mm^2 in a 0.5 μm CMOS process. The intention was to build a chip of 64 PEs on a 1.5 x 1.8 cm die achieving 2.56 billion calculations per second. The multiplier is noted as being the limiting factor on processor speed.

Two example algorithms are given for mapping to the architecture. The first is the feed forward operation of a neural network and the other is the implementation of a Kohonen self-organizing system. Both of these lend themselves very well to an SIMD machine.

While the performance numbers for this chip are impressive, again it must be noted that a purely SIMD machine is not an attractive candidate for reconfigurable computation. Mainly due to the lack of partial reconfiguration abilities as well as the lack of diversity in the computations between processors.

2.1.4.4 Penn State Micro-Grained VLSI Signal Processor

The Penn State micro-grained VLSI signal processor is a fine grained system consisting of a mesh of identical computational cells called digit processors [17]. The digit processors operate in SIMD mode and can be masked so that only those desired will execute on a given clock cycle. A given digit processor consists of two 3-input multiplexers for generating functions of three variables connected to a 16-bit multiport memory. Additional hardware contains configuration information and connection topology information. They are interconnected in a nearest neighbor organization.

The interesting part of this interconnection network is that the digit processors can be partitioned into square blocks called word processors. Different sized word processors can be partitioned for different tasks, though it appears that all word processors existing on the chip at a given time must be of the same dimensions. Further, all word processors appear to operate from the same instruction during a given clock cycle.

As noted by Irwin, the architecture is very similar to the Connection Machine 1 [18] with the major difference being that the cells of the CM1 are much larger in size. As a result, many more of the digit processors may fit on a chip than those found in the CM1. A processor based on this concept was in fabrication at the time which consisted of a 32 x 32 array of digit processors. The chip was simulated at 25 MHz for the 1.2 μm CMOS MOSIS process.

Being an SIMD processor, this entry does have limited reconfigurable computing potential, though some of the mesh aspects of the architecture are similar to the Colt chip, with the exception of the Colt's Skip Bus. The partitionable design scheme of the mesh is a valuable feature; however, and is noteworthy for that reason.

2.1.4.5 DPGA

The concept of a Dynamically Configurable FPGA has been proposed by Bolotski, et al. [19][20]. The idea is to allow context swapping within an FPGA. In a typical FPGA, the logic

---

17 Irwin, Mary Jane and Owens, Robert Michael, "A Micro-Grained VLSI Signal Processor," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. v-641 - v-644, 1992.
18 Hillis, W. D., *The Connection Machine*, Cambridge, MA, MIT Press, 1986.
19 Bolotski, Michael, DeHon, Andre and Knight Jr., Thomas F., "Unifying FPGAs and SIMD Arrays," MIT Artificial Intelligence Laboratory, Transit Note 95, September 1993.

functions are implemented as small RAMs or lookup tables. The logic inputs to the function are decoded and the logic value for that set of inputs is retrieved from a bit cell. Some of the configuration bits for the FPGA are used to set the values of these storage cells and these remain constant over the lifetime of a given configuration of the FPGA. The DPGA concept is to program several different sets of possible values for each logic function. The appropriate set could be selected at run-time using global signal wires to select the appropriate context at any given time. This would mean that the logic values for the function would be taken from a small RAM, and the global context control wires would act as the address used for that RAM.

In a later paper, DeHon advocates placing DPGAs on the same die as a normal processor to act as a reconfigurable accelerator [21]. Different configurations can be loaded into the DPGA and multiplexed at run-time to provide special purpose units to suit the current computational task. Thus, the configurations could be switched to match different applications being multitasked on the same processor or for different sections of the same application. Since the configurations are readily available and can be selected almost instantly, the current configuration could change from instruction to instruction.

One tradeoff that the DPGA concept makes is choosing reconfigurability over computational density. One can argue that the silicon area needed to implement the context switching RAM, decoding hardware and global control lines could be better used to implement

---

20 DeHon, Andre, "DPGA Utilization and Application," MIT Artificial Intelligence Laboratory, Transit Note 129, September 1995.
21 DeHon, Andre, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, California, pp. 31-39, 1994.

more computational cells. If the area cost of adding extra contexts is greater than adding more cells, then the choice should be in favor of more cells. Even if the area cost of adding more configurations is less than the cost of adding an equal number of cells (as seems likely for large numbers of configurations), then lost parallelism must also be considered. Part of the speedup gained by FPGA architectures is due to the extreme amounts of parallelism possible due to the large number of computational elements. If these elements are sacrificed in order to gain this type of context switching then there must be a break even point for speedup gained due to contexts versus speedup lost due to lack of parallelism. Having designed the Colt chip and seen that data storage comes at a high cost, a good case can be made for more computational elements based on a size argument. In any event, they can be multiplexed and used in the same context switching manner if desired. Further, the advantages of Wormhole Run-Time Reconfiguration allow a great increase in configuration I/O bandwidth over current reconfiguration schemes. Using the shorter reconfiguration times, it could be argued that the time to load new configurations onto a chip has become such that it is no longer necessary to consider storing them on the chip itself.

2.1.4.6 CAM DPGA
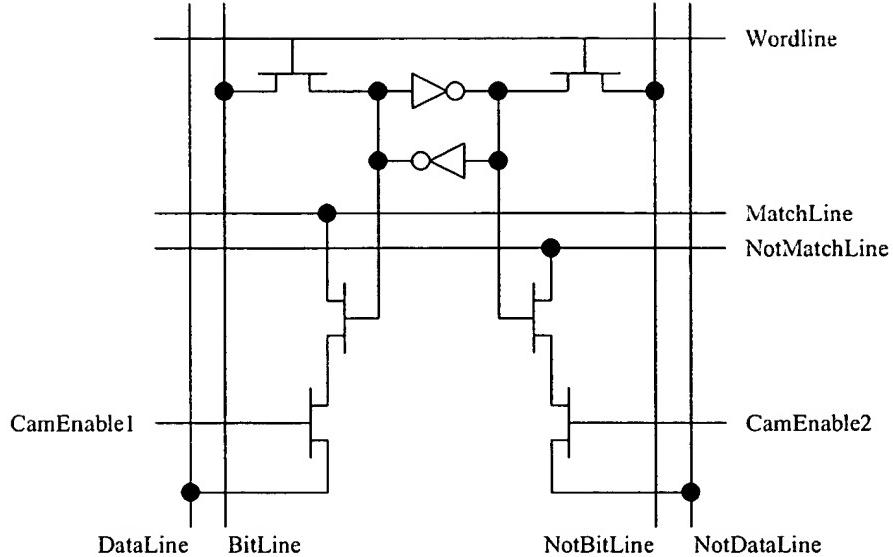
Figure 2.6 - CAM DPGA Cell.

The name Dynamically Programmable Gate Array (DPGA) has been overloaded in at least one source to infer the same type of architecture as a standard FPGA. The Content Addressable Memory (CAM) DPGA [22] has been proposed as a replacement for FPGAs that are based on the more standard lookup table based computational cells. Each cell contains a 4x4 block of Content Addressable Memory bits, giving the design its unique character. Four bi-directional connections are to be routed from each side and are connected to the four nearest neighbor cells. The four interconnection wires can be multiplexed to connect to various points in the 4x4 block of CAM cells. The basic CAM cell is shown in Figure 2.6.

22 Page, Ian, "Reconfigurable Processor Architectures," *Microprocessors and Microsystems*, vol. 20, no. 3, pp. 185-196, 1996.

The authors also discuss various methods for coupling a CCM architecture to a more conventional computational core, such as a microprocessor, and develop a loose taxonomy based on this. By using the reconfigurable properties of the CCM to create coprocessors for the core, a speedup in overall computation can be achieved. In these capacities, they argue that CAM cells are better suited to implementing the logic blocks commonly found in microprocessors, such as decoders and cache, than lookup table cells.

While the concept of using an array of CAM cells as the basic unit of computation does appear to have advantages, the authors do not address any issues with regard to the reconfiguration properties of the device. Neither the speed of reconfiguration, nor the control model used is discussed in their taxonomy of CCM devices. These issues must be encompassed by any taxonomy of CCM devices in order to fully classify any CCM device. Finally, the essentially bit-oriented processing of data used by the proposed device has unsavory implications for the number of configuration bits required for programming, which would adversely affect the reconfiguration time.

2.1.5 Systems

Though the thrust of this dissertation is to describe a chip-level architecture that is well suited to DSP type applications, many of the features of the resulting architecture are applicable at the system level. Also, the amount of useable silicon area available in the future will allow implementation of architectures at the chip level that are only possible in a full scale system today. Lastly, it is helpful to see the types of systems that people are building today to handle

DSP type tasks. For these reasons, a brief survey of some systems built for research purposes seems appropriate.

2.1.5.1 Splash 2

Splash 2 is a system based on Xilinx XC4010 FPGAs [23]. The system consists of a back plane into which up to sixteen processing boards can be inserted. Each processing board contains seventeen XC4010 chips, one of which is used to control routing between the others. The routing on a board is a limited crossbar configuration, limited in that not all possible connection permutations are possible. A number of interesting applications have been developed for Splash 2 including high speed image processing [24] with remarkable speedups over conventional implementations. However, the computational core of the Splash 2 system is the XC4010. As discussed above, the XC4010 does not support a number of features that are attractive for reconfigurable computing, including partial reconfiguration. Further, the slow reconfiguration time of the XC4010 further limits the abilities of Splash 2 as compared to a system based on the Colt.

---

23 Buell, Duncan A., Arnold, Jeffrey M. and Kleinfelder, Walter J., *Splash 2 - FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996, pp. 10-18.
24 Buell, Duncan A., Arnold, Jeffrey M. and Kleinfelder, Walter J., *Splash 2 - FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996, pp. 141-165.

2.1.5.2 Cheops

The Cheops system being developed by at the MIT Media Laboratory is a system designed to explore algorithms for image compression and processing [25]. Cheops is a modular system consisting of three component types: Input Modules, Output Modules and Processing Modules. Input modules are designed to digitize video input data and do limited preprocessing. Up to four of each type can be plugged into a system back plane. Output modules take digitized data as input and output analog video in several different formats. One of the constructed processing modules, called a P2, contains an Intel 80960CF RISC processor that can achieve two instruction per clock cycle performance at 32 MHz [26]. In addition, up to eight daughter board connections are included for incorporating coprocessors, such as FIR filters, transform engines, correlators or an entire custom subsystem designed to perform a specialized task. Six of these daughter boards are on removable submodules. These units are connected to a large local memory, a SCSI interface, RS-232 port, a AES/EBU interface for digital audio and an interface to the global bus. A 17x17 crossbar connects all the resources on a processor board with a 16-bit data channel running at 40 MHz. Each processor board can be accessed and controlled through the SCSI interface, providing an interface to the host that makes the board look much like a disk drive.

Communications at the system level are handled by a global bus and two Nile buses. The global bus acts as a command channel for synchronizing system events, programming DMA

---

25 Bove, Jr., V. Micheal and Watlington, John A., "Cheops: A Modular Processor for Scalable Video Coding," *Proceedings of SPIE*, Bellingham, Washington, vol. 1605, part 2, pp. 886-893, 1991.

controllers, etc. Each board in the system is given an address range on the global bus, connecting the entire system in a shared memory configuration, transferring 32-bit data at the rate of 16.67 MHz. The Nile buses are intended as the main path for video data through the system. Each is 48 bits wide, running at a 20 MHz clock rate. Transfers over a Nile bus are handled by DMA controllers that are programmed through the global bus.

The real power of the machine comes from the ability to plug in special purpose daughter boards that can perform any function desired. One such board is called the Splotch Engine [27]. The Splotch accelerator board is used as a coprocessor for Cheops that allows it to act as a 3D hologram generator using Hogel-Vector encoding. The image generated is approximately 100 mm on a side and the 36 megabytes of display data necessary is generated in 3 seconds. In their conclusions, the authors cite a lack of communications bandwidth offered by the Nile buses as a future bottleneck to possible speedups. The system could be sped up by cascading processor boards, but then the problem becomes one of moving data between the boards and to the output driver at the same time. This type of problem would not exist in a Wormhole RTR system since there need not be a concept of a global data bus for moving large amounts of data through the system. The Wormhole RTR concept, as described in Chapter 3, would alleviate the bottleneck by eliminating the global control concept introduced by a common bus used for data transfers.

---

26 Intel Corporation, *80960CA User's Manual*, Santa Clara, CA, Intel, 1989.
27 Watlington, John A., Lucente, Mark, Sparrell, Carlton J. and Bove, Jr., V. Michael, "A Hardware Architecture for Rapid Generation of Electro-Holographic Fringe Patterns," *Proceedings of SPIE*, Bellingham, Washington, vol. 2406, pp. 172-183, 1995.

Another interesting use of the daughter board connections is a run-time reconfigurable board called the State Machine [28]. The State Machine consists of two Altera FLEX EPF81188 FPGAs and a PowerPC 603 chip. Two data stream ports come onto the daughter board from the crossbar, one to each of the Altera chips, each of which is also connected to 64K of look up table and 256K of normal SRAM. The FPGAs can be reconfigured by the processor board 80960, tailoring them to whatever task is needed. At the time of writing only a few applications had been written to take advantage of the State Machine, one of which is the evaluation of the optical flow vector equations:

$$dx = Ax + By + C$$

$$dy = Dx + Ey + F$$

The x and y values come into the daughter board and generate a (dx,dy) pair every two clock cycles, with the data then being forwarded on the crossbar connections. The authors note that the routing constraints of the Altera FLEX chips do not allow efficient mapping of pins to an arbitrary configuration. Because the pin connections are hardwired at the daughter board level, no design had been able to utilize more than 46% of the available resources while still being able to access the required pins.

The Colt chip and the stream concept to be presented later have at least two advantages over this system. The first is that the Colt chip routing resources allow any arbitrary port to be connected to any I/O point in the device, alleviating the board level pin out problems described.

---

28 Acosta, Edward K., Bove, Jr., V. Michael, Watlington, John A. and Yu, Ross A., "Reconfigurable Processor for a Data-Flow Video Processing System," *Proceedings of SPIE*, Bellingham, Washington, vol. 2607, part 2, pp. 83-91, 1995.

Further, the Cheops system uses a global controller to program the FPGA devices, which has many disadvantages compared to the distributed programming scheme offered by streams as described later.

2.1.5.3 Teramac

Teramac is a reconfigurable system built by Hewlett Packard [29]. The system consists of 1,728 custom FPGAs, mounted in Multichip Modules (MCMs) in groups of 27. There are 4 MCMs per board and up to 16 boards per system. The exact details of the routing between the components seems somewhat mysterious, and the exact architecture seems to be something of a company secret. However, some detail is known about the custom FPGA known as PLASMA. It consists of an array of 6 input, 2 output lookup tables, which are connected by a partial crossbar. There are 336 pins useable for I/O purposes, and there are enough routing resources available on chip to make design placement and routing independent of the desired pin out. Each lookup table has configurable latches and registers at its outputs. Routing resources were made a priority to keep the placement and routing cycle to a reasonable amount of time. The authors point out that even 10 minutes per chip to place and route a design on a standard FPGA would have taken 12 days with 1,728 FPGAs. The machine contains 0.5 Gigabytes of RAM and can be clocked at up to 1 MHz.

Several test applications had been compiled for Teramac, including a bubble sort, graph partitioning and the calculation of Pi. Two more interesting applications that have been

51

developed are an Artery Extraction Filter and Cube, a volume rendering system [30]. The Artery Extraction Filter takes a volume of magnetic resonance imaging (MRI), 256 points on a side and attempts to determine if each point is the center of an artery. If so, it calculates an ellipse that encircles it. This application achieves a factor of 110 speedup over a standard processor and runs at a clock rate of 650 KHz. Cube renders volume data, shading surfaces and creating cut away views. This application runs at 920 KHz and achieves a speedup of 2.76 over a workstation simulation. Unfortunately there is not enough information about Teramac to really make comparisons, but it would be remiss to at least not mention the existence of such a large machine.

2.2 Data Driven vs. Control Driven Concepts

From the speed computations above it seems obvious that the system needed must employ some form of parallelism to accomplish the computational goals. The types of computations that are involved with DSP type applications, such as the implementation of an FIR or Adaptive Filter, suggest a data flow approach. The impetus for moving to a data driven architecture is to obtain optimal instruction execution ordering and maximal parallel instruction execution under all circumstances. The degree to which these two goals can be attained is a function of the program and the data set.

---

29 Amerson, R., Carter, R., Culbertson, B., Kuekes, P. and Snider, G., "Teramac - Configurable Custom Computing," *Proceedings of the 1995 IEEE Symposium on FPGA's for Custom Computing Machines*, 1995.
30 Culbertson, W. Bruce, Amerson, Rick, Carter, Richard J., Kuekes, Philip and Snider, Greg, "Exploring Architectures for Volume Visualization on the Teramac Custom Computer," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

Data flow programs avoid many of the instruction scheduling problems associated with control flow programs by explicitly specifying all possible parallelism and data dependencies in the program. This allows the execution of a data flow program to proceed at the fastest possible rate by taking advantage of the maximum amount of parallelism possible within the limits of a given program and available hardware. Hence, a data flow program can be easily scaled up into a larger hardware platform containing more processors by dynamically and automatically expanding to take advantage of more of the inherent parallelism. The program does not even have to be recompiled. Such capabilities are difficult to achieve on a control driven machine while on a data flow machine they are inherently present.

Data Flow Graph

Control Flow Program

Step	Instruction
(1)	$t_1 = a_1 * x$
(2)	$t_2 = a_2 * x$
(3)	$t_3 = a_3 * x$
(4)	$t_4 = t_1 + t_2$
(5)	$f(x) = t_3 + t_4$
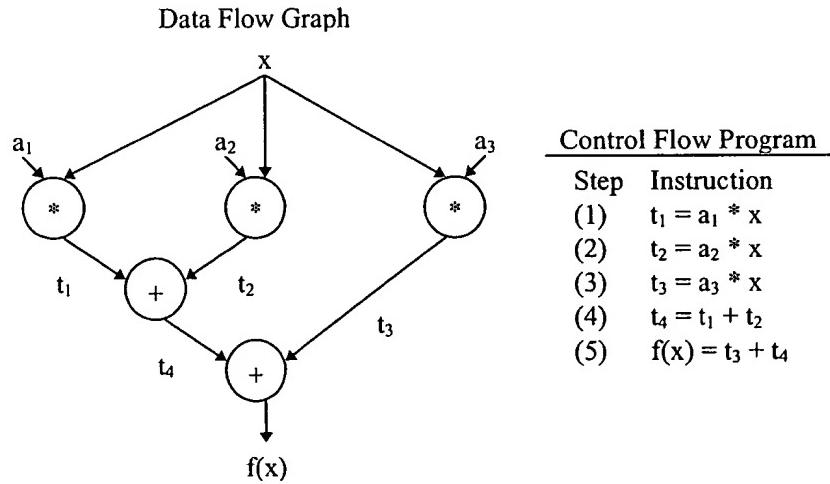
Figure 2.7 – Data Driven vs. Control Driven Example.

Figure 2.7 shows an example data flow graph that evaluates the function $f(x) = a_1{}^*x + a_2{}^*x + a_3{}^*x$. It resembles a familiar flow chart in many respects and conveys similar information. Note that there are four inputs to the data flow graph; x, a_1, a_2 and a_3. The top three nodes in the graph represent the multiplication operator and the lower two show the addition

operator. The connecting arcs in the graph represent the flow of data. A particular node in the graph cannot be evaluated until all required inputs are available. Once that condition is satisfied it can be evaluated at any time. For example, the left addition node requires t_1 and t_2 to be available before output t_4 can be produced. A data flow program contains the encoded connecting arc information in some form along with the operations to be performed. A data driven machine must have a system for processing the dependencies indicated by the arc information to keep track of when the inputs for each node are available and then scheduling those nodes for execution.

The control driven program in the right of Figure 2.7 accomplishes the same calculation using instructions that are available in a typical control driven processor. Note that the order of execution for the operations is specified by the order that they are listed, which is typically the order that they appear in memory. Control flow programs contain no explicit information indicating which operations produce the inputs for the next instruction. It is assumed that the operands for a given instruction are ready when the machine reaches it. Thus, a control driven machine needs no special system to store the operand availability of every operation in the program. For this reason, less hardware and less memory are needed in a simple control driven machine. However, the lack of such information also makes it very difficult to build a control driven machine that can execute these instructions in any order other than that laid out in the program. The method by which the seemingly simple requirement of proper ordering of the operations is maintained is what differentiates data driven programming from control driven programming.

2.3 Data Flow vs. Control Flow Tradeoffs

The program specifies all dependencies between data items; explicitly if it is specified in data flow graph form, implicitly if it is specified as a control flow program. These dependencies dictate the minimal set of restrictions on execution order that will still guarantee a correct result. For example, a single processor control driven machine executing the code in Figure 2.7 would generate the products one at a time and then perform the two additions. In a parallel system all three products can be done simultaneously and then the two additions would be performed, thus increasing the speed of execution by nearly a factor of three since multiplication usually takes much longer than addition. A control flow version of the program to implement this graph only contains implicit dependency information in its program specification. Hardware in the form of reservation stations, scoreboarding or other techniques are required in a control driven computer to determine these data dependencies at run-time [31]. These units track operations in progress along with data dependencies between operations and attempt to dynamically schedule new operations as quickly as possible given the constraints imposed by data and hardware availability. The difference in scheduling between the control driven approach and data driven machines is that the data flow program explicitly contains all the dependency information shown in the picture, thus simplifying the task of determining which operations can be executed in parallel.

---

31 Smith, J. E., "Dynamic Instruction Scheduling and the Astronautics ZS-1," *IEEE Computer*, vol. 22, no. 7, pp. 21-35, 1988.

Data Flow Graph

Control Flow Program

if (x < 10)
    f = y + z;
else
    f = w + z;
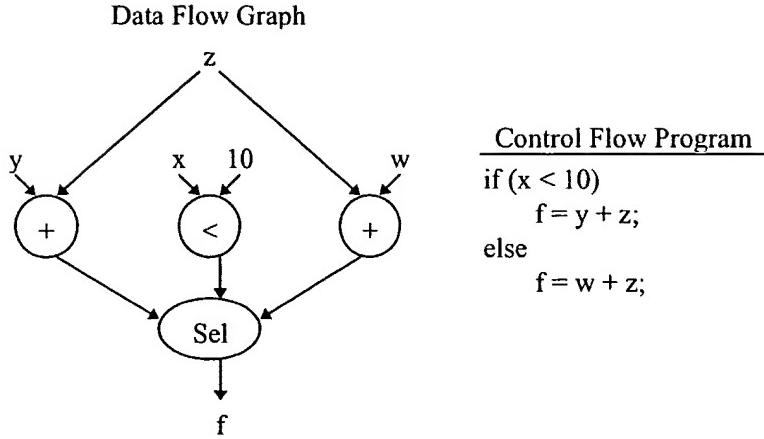
Figure 2.8 - Example Of Data Dependent Execution.

However, data dependencies are not necessarily the same every time a given program is run. Conditional execution within a program can affect what data items are required to generate the result needed for a given instruction to run. For example, in the code shown in Figure 2.8 the variable f will require operands y and z if x is less than 10. However, if x is greater than or equal to 10 it will need w and z. The calculations needed to generate w, y and z will be different and the inclusion or exclusion of the code to generate these intermediate values will affect the optimal scheduling of overall program execution.

Another factor involved in optimal scheduling of a program on a parallel machine is instruction execution time. If all instructions execute in a fixed amount of time then there is no problem. However, if instructions have variable execution time, perhaps dependent on an input operand, optimal scheduling at compile time becomes impossible.

2.4 Recent Control Flow Trends

Many control driven computers today attempt to mimic data flow computers to various degrees. Superscalar computation can be considered to be a data flow concept in that the same problem of matching available results with operations arises. In a control driven processor, the hardware must extract data dependencies from the instruction stream at run-time to determine what data items are required before a given operation can be performed. As stated above, the program for a data driven computer contains these dependencies explicitly.

Another architectural trend of control flow processors is out-of-order instruction execution. Superscalar computation is usually coupled with out-of-order execution, which can be defined as performing instructions in an order other than that specified by the programmer. This can be performed in a superscalar machine when a multiplication (long execution time) is executing, but it is followed by a non-dependent addition (short execution time) in the instruction sequence. In a superscalar machine, the addition can be issued and executed in parallel with the multiplication.

An example of a control driven processor that has been taken to extreme steps to become more data flow like is the Intel Pentium Pro (P6). The P6 employs a number of methods that Intel collectively calls Dynamic Execution [32]. The processor is built around a small memory pool of micro operations. A fetch and decode system reads in the 80x86 instructions from memory and decodes them into micro operations. At the same time, the registers referenced by

---

32 http://pentium.intel.com/procs/p6/dynexec.htm

57

these instructions are re-labeled with tags that represent the data in the registers rather than the registers themselves. The re-labeled micro operations are then put into the pool of micro operations. A superscalar computational core of degree 5 (5 operations can be issued in parallel) looks through the pool of waiting micro operations and issues those whose tagged data and execution unit are available. As results come out of an execution unit (integer addition for example), they are put back into the pool. If these results were needed by another micro operation the second operation can then execute. In any event, the finished operation is taken out of the pool and "retired" in the order of the original instruction stream by updating the actual register set of the processor. It is important that instructions are retired in order on a control driven processor, because the programmer depends on that order.

In effect, the P6 is designed to read in a control driven program, create a data flow representation of it internally, execute it as such, and then output results that mimic those of a control driven processor. The amount of effort required to realize this data flow architecture is greatly complicated by the hardware necessary to convert the control flow code into a data flow representation. Intel's motivation for maintaining their support for the control driven 80x86 instruction set standard is obviously driven by mass market appeal considerations. However, equally obvious is their recognition that data flow computers have inherent advantages over control driven architectures. Intel's next major entries into the market place will no doubt continue to support the data flow paradigm.

2.5 Data Flow Architecture Classifications

It is appropriate to establish some of the taxonomy of data flow computers. There are two major classifications of data flow architectures based on the flexibility of operation execution as specified by DeCegama: static and dynamic [33]. In an excellent survey paper by Veen, data flow computers are further classified by type of communication that exists between processing elements: direct and packet communication [34].

2.5.1 Static Data Flow Architectures

In static data flow architectures, it is assumed that two operands will not occupy the same arc in the data flow graph at the same time. Hence, there is no possibility of reordering the operands by accepting the wrong one from the incoming arc. To guarantee this condition, acknowledgment arcs are added to the data flow graph. The receiving operator generates a token that is sent to the sending operator to indicate when the previous data value has been consumed. This increases the amount of token traffic by a factor of 1.5 to 2, increasing the demands on the hardware and causing performance to suffer. Also, exploitable parallelism is lost from the program due to the full handshaking mechanism.

---

33 DeCegama, Angel L., *Parallel Processing Architectures and VLSI Hardware*, Englewood Cliffs, NJ, Prentice Hall, 1989, vol. 1, pp. 146-157.
34 Veen, A. H., "Dataflow Machine Architecture," *ACM Computing Surveys*, vol. 18, no. 4, pp. 377-379, December, 1986.

2.5.2 Dynamic Data Flow Architectures

A dynamic data flow architecture is created with the assumption that many tokens may exist on a given arc at the same time. The problem of matching the correct pair of operands then arises as there may be several operands on each incoming arc of a consuming operator. These operands may be part of the same computation at different time steps or they may be part of different computations altogether. Matching of operands is accomplished by assigning a tag to each operand that consists of an invocation ID, an iteration ID, a code block and an instruction address. The invocation ID and iteration ID indicate which node in the graph produced this result and when. The code block and instruction address indicate the destination for the result of this operation.

In a dynamic data flow architecture, data operands enter into the computational core and the tags are used to matched them up with the operator that needs to be applied to them. Data moves through the machine as fast as it can be processed. An operation does not wait for another to finish if there are no data dependencies between them. Hence, if two independent tasks are being performed in a loop, and task A is faster than task B, then task A could be on its fifth iteration while task B is still on its second. The lock step execution of a normal single processor control flow architecture prevents this type of loose coupling of data. On a data flow architecture, if a given operator has a complete set of operands that match it can be executed; the specification of the data flow graph guarantees that no dependency problems exist. In a dynamic data flow system, the dependencies specified by the data flow graph are enforced by tags, which act as the scheduling semaphores of the machine.

2.5.3 Direct Communication Architectures

Veen defines a direct communication architecture as a machine in which adjacent operations in the data flow graph are implemented using the same processing element or on directly connected processing elements. He further specifies that the order of operands in such a system is maintained so that tags are not needed to match up operands. The arcs of the graph behave as FIFO queues.

2.5.4 Packet Communication Architectures

Veen defines a packet communication architecture as one in which operands are sent as packets of information through an unknown channel to their destination processing element. The latency of the communication channel may vary from point to point, so that it is possible for operands to arrive out of order at the destination node. Typically, tags are needed in such a system. However, he also points out that this type of system is capable of the greatest parallelism and best load distribution since bottlenecks do not occur on a particular channel. Parallel channels of communication can be added to improve throughput, though many new problems then arise with network congestion and the possibility of deadlock.

2.6 Data Structures For Data Flow

As defined by DeCegama, there are three types of data structures that are used in programming a generic data flow computer: scalars, streams and arrays [35].

2.6.1 Scalars

A scalar is essentially the same as the mathematical definition implies. A single piece of information that travels through the system autonomously. These do not promote great efficiency in terms of pipelining.

2.6.2 Streams

A stream as defined by DeCegama is a data structure on which production and consumption are done in a sequential pattern. No random access operations are permitted.

2.6.3 Arrays

An array is a data structure that can be accessed randomly. This is a problem in a data flow architecture since the access pattern is not known at compilation time and no real notion of an address exists. Further, the basic tenant of data flow dictates that the inputs to a given operator are not changed by the operator. A new operand is simply produced. In order to simulate array accesses, two data structures have been proposed: heaps and I-structures. Both

---

35 DeCegama, Angel L., *Parallel Processing Architectures and VLSI Hardware*, Englewood Cliffs, NJ, Prentice Hall, 1989, vol. 1, pp. 157-160.

methods essentially keep a list of pointers to the array items that can be dereferenced to access the actual data values. The data items themselves are not changed, and new lists of pointers can be created as necessary. The heap method sorts the pointers based on branch and leaf numbers in a heap. Additions to the heap are pushed down from the root to the appropriate leaf node. The I-structure approach maintains a list of slots that are filled as data becomes available. A state is maintained for each slot to indicate whether or not it has received data. Also, a list of pending requests is maintained so that when the data for a particular slot is produced, it can be forwarded to all waiting consumers.

3. Solution Approach

In this chapter, the basic approach to solving the posed problems will be delineated. First, the type of data flow computation that can be implemented on Colt is discussed, touching on various issues with hardware tradeoffs that were made. The concept of virtual hardware will then be discussed as it is concerned with CCM devices. Finally, Wormhole Run-Time Reconfiguration will be introduced and discussed at length as the chapter concludes.

3.1 Data Flow Implementation

As was discussed in Chapter 1, FPGA-type architectures lend themselves to the direct hardware implementation of data flow graphs. Specifically CCM devices can be used to easily implement Direct Communication Static Data Flow (DCSDF) architectures as defined by Veen [36]. The use of tags is surperfluous in DCSDF architectures because tags are largely required to deal with the problems of synchronization, order preservation and other problems of reentrant data flow graphs. The definition of the DCSDF architecture disallows such graphs. Tags were briefly described as being the encoding that prevents operands from being reordered and mismatched as the execution of the machine proceeds. It would seem that a machine constructed without tags would suffer from synchronization and operand ordering problems; however, if

some restrictions are introduced into the execution patterns in the machine, tags may be ignored while keeping many of the benefits of a data flow architecture. The impetus to remove tags stems from the hardware expense of implementing a tagged system. One aspect that discourages implementation is the large associative memory required to match the tag of one operand to another and to match those to the appropriate operator. This type of overhead is similar to that of a cache in a control flow processor. Another resource intensive aspect is the routing requirement of shuttling the data between execution units and memory. If an alternative solution were used, these resources could be used for the creation of additional computational units.

One of the drawbacks of a tagless data flow machine is that operand ordering must be strictly preserved as the computation proceeds. In a tagged architecture, the temporal information is stored in the tag, but because the temporal ordering is not stored in a tagless machine, the operands must be queued along the arcs of the data flow graph. Execution proceeds at the pace of the slowest computation in the pipeline. Also, since a tagless machine stores no temporal information with each data item, the total length of any two execution paths that converge on a single node must be exactly the same length in terms of execution time. Consequently, the execution time through each node in the data flow graph must be precisely determined so that the appropriate pair of operands arrive at a given node at exactly the same clock pulse. This restriction seems to cripple processing by increasing the execution time of the entire graph to that of the longest path; fortunately that is not the case. By breaking the data flow graph into smaller subsections, or *pages*, this critical timing criteria need only be applied to all

---

36 Veen, A. H., "Dataflow Machine Architecture," *ACM Computing Surveys*, vol. 18, no. 4, pp.

computational paths within a given page. The number of nodes contained in a page is limited by the available hardware resources. By choosing page boundaries so that convergent computational paths with large differences in latency occur in different pages, the resource burden of delaying the shorter path may be avoided.

For large computations, only small sections of the data flow graph may be executing simultaneously, even on a massively parallel machine, due to the complexities (in terms of VLSI hardware, etc.) of the calculations involved. As a result, there is a need to store intermediate computations. For example, if a single multiplier were being used to perform a dot product of two three-element vectors in a given data flow machine, then three intermediate results may need to be stored as that multiplier were used for each successive operation. Since the data flow graph must be executed in relatively small subsections, producing intermediate results that are then fed into other subsections, some type of storage is required for these intermediate results. This storage may be provided by a RAM in some cases.

Since memory is already needed, the memory could also be used as a synchronization area where intermediate results for paths that have different execution lengths could be stored. Intermediate operands can be queued in memory as they are produced. When all inputs that are required to execute a given piece of the graph are available, the subsection can be scheduled for execution. A subsection of the data flow graph that can be directly implemented on the system will be defined as a *page*. A page can consist of one or more operator nodes. All paths within a page must satisfy the timing equalization goal: any two execution paths that converge on a single

---

377-379, December, 1986.

66

node must have the same path length in terms of clock cycles of execution time. Unequal paths can be equalized by inserting delays into the shorter execution paths to make them match the latency times for the longer execution paths. Hence, the graph resembles a programmable pipeline.
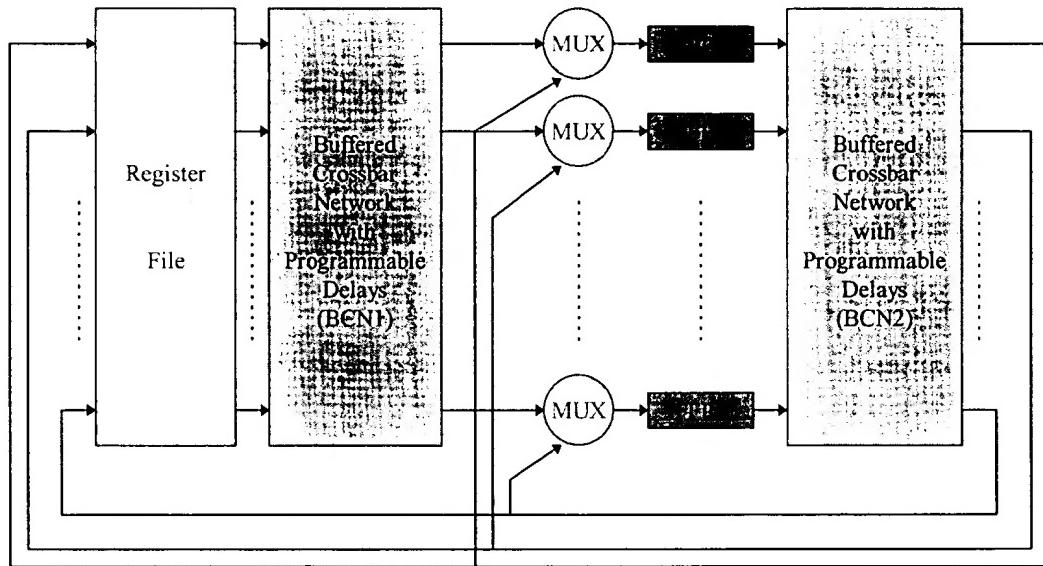
Figure 3.1 - Pipelined Net Architecture [37].

The concept of a pipeline with a reconfigurable interconnect has already been dubbed a *Pipelined Network* or *Pipenet* by Hwang and Xu [37]. They describe the use of Pipenets for the computation of Complex Vector Functions (CVFs) in scientific computing. Figure 3.1 shows the basic Pipelined Net architecture in which several Functional Pipelines (FPs) are connected through dual crossbars so that the output of any FP can be routed to any register and from there to any other FP. Multicasting and broadcasting are also possible using the crossbars. Further, the

---

37 Hwang, Kai, *Advanced Computer Architecture*, New York, NY, McGraw-Hill, 1993, pp. 442-446.

crossbars act as variable delay units to overcome the problem of equalization of the execution path length discussed above. Hwang and Xu chose a full crossbar interconnection scheme over multistage packet switching networks due to the need for arbitrary connections when mapping various CVFs to the Pipelined Network [38]. The FPs are intended to perform different operations at different times, and are themselves pipelined for higher throughput. Because of this, the authors refer to the system as having two level pipelining. The use of the system was envisioned by the authors as having a supervising processor attached to the Pipenet structure that uses special instructions to reconfigure the routing network, issue instructions to the FPs and perform job control. No systems based on the Pipelined Network concept have ever been built.

Other papers on the subject more fully explore the concept, including methods for mapping CVFs onto the architecture [39]. This includes a language construct for specifying the use of Pipelined Nets, example instructions for use in controlling the structure, algorithms for partitioning graphs for implementation and relative speedups of the approach. Lastly, [39] illustrates examples of loops that can and cannot be implemented on the architecture from the set of Livermore Loops as developed by the Lawrence Livermore National Laboratory [40].

---

38 Hwang, Kai and Xu, Zhiwei, "Multipipeline Networking for Fast Evaluation of Vector Compound Functions," *Proceedings of the IEEE International Conference on Parallel Processing*, New York, NY, pp. 495-502, 1986.
39 Hwang, Kai and Xu, Zhiwei, "Multipipeline Networking for Compound Vector Processing," *IEEE Transactions on Computers*, New York, NY, pp. 33-47, 1988.
40 Riganiti, J. P. and Schneck, P. B., "Supercomputing," *IEEE Computer*, vol. 17, pp. 97-113, October, 1984.

Table 3.1 - Chebyshev Terms Required To Approximate Elementary Functions.

Function	Range	Mantissa Length (in bits)			
		24	32	53	64
e^x	[0,1/16]	3	4	6	8
Ln(x)	[1/2,1]	3	4	6	8
Sin(x)	[0,π/2]	4	5	8	9
Cos(x)	[0,π/2]	4	5	8	9
Tan(x)	[0,π/8]	4	5	8	10
$Tan^{-1}(x)$	[0,Tan(π/12)]	3	5	8	10

Hwang, et al., also described how Pipelined Nets can be used to calculate approximations to elementary functions using Chebyshev Polynomials [41]. They categorize elementary functions as exponential, logarithmic, trigonometric, inverse trigonometric and other transcendental functions. Table 3.1 shows their results for the number of approximation terms required to achieve acceptable precision for the various mantissa lengths shown for several functions. These correspond to the lengths used in the IEEE floating point standard for single, single extended, double and double extended formats [42]. They present an in-depth description of how the approximation can be implemented efficiently on the Pipenet architecture. These optimizations include range reduction which takes advantage of periodicity, symmetry and recurrence relations to calculate the elementary functions. These results all directly apply to the Colt CCM because of the inherently data flow implementation style that the calculations use.

---

41 Hwang, Kai, Wang, H. C. and Xu, Z., "Evaluating Elementary Functions with Chebyshev Polynomials on Pipeline Nets," *8th Symposium On Computer Arithmetic*, New York, NY, pp. 121-128, 1987.

42 IEEE Press, *IEEE Standard 754 for Binary Floating-Point Arithmetic*, New York, NY, 1985.

Remps is a system based on the Pipelined Network architecture that has been proposed by Hwang, et al. [43][44]. The Remps supercomputer was envisioned as being composed of many Pipenet structures, all connected to a shared memory and I/O subsystem. These are orchestrated by a global controller that issues instructions to allocate tasks to individual Pipenet structures. The local controller in the Pipenet then forms a complete pipeline from the memory through the Pipenet structure and back to memory again and begins processing. This structure is called a *macro pipeline*. Pipenets can also be cascaded so that the intermediate step of sending data to memory does not need to be performed. The proposed uses of the Remps machine are scientific computing, parallel PDE solutions and performance analysis of benchmark programs.

Probably the most intriguing machine proposed by Hwang, et al., is Opcom [45]. Opcom is a proposed digital optical computer based on the Pipenet concept. In this machine, an array of optical cells are connected to an optical crossbar. The cells can perform the logical NOR operation and hence are logically complete. They can also act as single bit storage elements. A control unit is connected to the host and from there the crossbar network is configured into the correct Pipenet pattern. Data is copied into the cell array, and then processing proceeds at optical speeds. The controller is very simple, utilizing a small set of instructions to make connections

43 Hwang, Kai, Xu, Zhiwei and Louri, Ahmed, "Remps: An Electro-Optical Supercomputer for Parallel Solution of PDE Problems," *Proceedings Of The 2nd International Conference on Supercomputing*, St. Petersburg, Florida, pp. 301-310, 1987.
44 Hwang, Kai and Xu, Zhiwei, "Remps: A Reconfigurable Multiprocessor for Scientific Supercomputing," *Proceedings of the IEEE International Conference on Parallel Processing*, New York, NY, pp. 102-111, 1985.
45 Xu, Zhiwei, Hwang, Kai and Jenkins, B. Keith, "Opcom: An Architecture for Optical Computing Based on Pipeline Networking," *Proceedings of the 20th Annual Hawaii International Conference on System Sciences*, Vol. 1, pp. 147-156, 1987.

between two arbitrary cells in the array and for basic looping and branching operations through a small program. The controller can start and stop the array using conditional logic and then can be used to reconfigure the connection pattern. A halt instruction is issued to the host when processing has been completed.

No systems have been built around the Pipenet concept and these two proposed systems are the extent of the published work on the topic. The Colt CCM is an embodiment of these ideas. Being a different architecture in terms of the interconnection resources that are available, Colt lacks the benefits of the full crossbar concept. Many sacrifices were made with regards to the ideal system architecture as the realization of Colt came to fruition because of the restrictions of a real-world implementation became apparent. Nonetheless, the heart of the computation paradigm used on Colt has its roots in the Pipenet concept.

3.2 Virtualized Hardware

The concept of implementing data flow graphs directly in hardware using programmable logic such as FPGAs is not new [46]. Neither is the concept of virtualizing hardware resources using reconfigurable technology [46,47]. Virtual hardware is the idea that large hardware systems can be emulated using a small piece of configurable hardware that can be rapidly programmed to function as any given part of a larger hardware platform. Since different

---

46 Ling, X. P. and Amano, H., "WASMII: a Data Driven Computer on a Virtual Hardware," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 33-42, April 1993.

computation paths generally require different execution times, the creation of a large hardware platform that embodied all possible paths could result in low utilization of the various components. By using a small programmable core to simulate the larger system, the utilization of the core can be kept higher as it is reprogrammed to perform the functions of the various sections of the larger platform. When one section becomes idle, another section can be programmed and processing can continue from that point. Further, the area and cost required to implement the larger system can be saved because the core emulator is by definition smaller. The cost of a virtual hardware implementation can be measured in terms of mapping efficiency, storage requirements and reconfiguration time. It seems obvious that the full implementation of the platform would attain higher execution speeds. However, it may be the case in a real-time system that processing speed outstrips the data acquisition rate. In these cases, a virtual hardware platform seems reasonable since the programmable emulation core may be able emulate the various parts of the full platform and complete processing within a reasonable amount of time. Indeed, this is in a sense the exact function that microprocessor systems fill in that the function performed by the microprocessor is changed with each instruction as the greater computation is performed.

Virtualized hardware is a concept that is exploited in the Colt CCM. The requirement of use and reuse of the same hardware to perform different tasks demands the use of configurable computing in the form of CCM-type devices. CCM devices offer the programmer the added capability of fine tuning the various units of the hardware to take advantage of algorithm

---

47 Casselman, S., "Virtual Computing and The Virtual Computer," *Proceedings of IEEE*

dependent optimizations. In most general purpose machines, specialized units are created in silicon to perform a certain set of functions very quickly. These functions are carefully chosen on the basis of predicted frequency of occurrence in the class of applications that is being targeted. Inevitably, only a subset of these resources are used when a designer applies the hardware to a specific task. Further, they are designed to solve the most general case of the functions, even though simplifying assumptions about their calculation may be known for a given algorithm, leading to wasted hardware and execution time. Both of these problems are addressed with the use of reconfigurable computing in conjunction with the concept of virtualized hardware.

Using the Analog Devices' SHARC chip described in Section 2.1.2.1 to implement the FIR Filter in Chapter 1 is the perfect example. The SHARC chip can do at least two floating point operations per clock cycle. The FIR filter requires only integer arithmetic, so all the extra silicon devoted to floating point effectively goes to waste for this application. A reconfigurable machine is designed so that the same hardware resources can be applied to a wide variety of operations that may be required by different algorithms. Run-time customization to the task at hand is one of the design goals of this system.

73

3.3 Wormhole RTR

3.3.1 Motivation

Contemporary CCMs rely upon RAM-based FPGAs as the mechanism of achieving reconfigurability. Even though these devices were not designed for computing, they have proven quite effective in demonstrating the potential for high-performance computing. One of the limitations of contemporary FPGAs is the reconfiguration mechanism. There are two primary approaches commonly adopted: serial configuration and random access configuration. In devices using serial configuration, such as the Xilinx XC4000 series [48] and the Altera FLEX 8000 line [49], the configuration storage elements are connected as a large scan chain around the entire chip. During configuration, the programming information is loaded into the device and shifted throughout in a bit-wise manner. While parallel loading of the configuration data is possible with some devices, close examination of the timing employed by these devices reveals an internally serial architecture. Most often, the entire chip must be programmed in such fashion before any part of it may be used to perform useful computation.

In a Run-Time Reconfigurable environment, the serial method of configuration has a few disadvantages. For one, the concept of partial reconfiguration is not supported. There are many occasions in configurable computing when it is advantageous to reconfigure part of a device

---

48 Xilinx Incorporated, *The Programmable Logic Data Book*, San Jose, California, 1994, pp. 2-4 - 2-43.

49 Altera Corporation, *Altera 1995 Data Book*, San Jose, California, 1995, pp. 37-93.

while leaving the rest intact, such as when changing the constant used in an active calculation. Also, the configuration/execute cycle must be done sequentially due to the fact that the entire device must be configured before any part may be used for execution. In a Run-Time Reconfigurable machine, this configuration/execute cycle may be compared to the Von Neumann fetch/execute cycle of a microprocessor. All high speed microprocessors of today overlap the fetch and execute cycles to some degree in an attempt to gain the speedups offered by a more pipelined approach. It would seem advantageous to barrow this concept and apply it to CCMs. Moreover, the scheduling of multiple independent processes onto a single device is also hampered by an all-or-nothing configuration scheme. The ability to configure one region of a chip while simultaneously executing within another region would aid not only in pipelining the configuration/execution cycle for a single process, but also in the multitasking of several processes onto the same chip. This concept will become increasingly valuable as the size and density of FPGA devices increases. Further, as more full-sized CCM systems are developed consisting of multiple devices, the ability to share the same set of resources across multiple processes, or different threads of the same process, will prove its worth.

Another deficiency of the serial configuration method is a lack of speed. The benefits gained from the use of a set of truly parallel configuration lines is intuitively obvious. In CCMs, speed of configuration often translates directly into performance, particularly when multiple configurations must be swapped in and out of a device in quick succession. Thus, it would seem desirable to widen the data path used for configuration information.

There are attempts to widen the configuration bottleneck by incorporating a truly parallel data path for programming information. The Xilinx XC6200 series [50] and the National Semiconductor CLAy series [51] use a random access method for reconfiguration. The configuration cells for these devices can be accessed in the same way as a standard RAM. An on-chip row/column address is presented to the device, and the programming information is either read from or written to the desired cells. This solves many of the problems associated with serially configurable FPGAs. Partial reconfiguration is supported, and, to an extent, configuration time is lessened through the use of an 8-bit configuration path for the CLAy, and a 32-bit configuration path for the XC6200. While these improvements are welcomed, there are still shortcomings. One fundamental limitation of the design is the implied used of a centralized control scheme. Serially configurable devices suffer from this downfall as well: only one controller at a time can configure the device through the access port. While access to that port can be time multiplexed, only one data path is used for configuring the device at any given time. Further, in terms of silicon area, the infrastructure needed to support the random access approach can be quite expensive due to the global scope of the routing and control logic required.

An alternative approach is to create a distributed control scheme in which multiple independent computational streams can configure the system simultaneously through multiple access ports. One advantage of a distributed scheme is scalability. The global controller in a

50 Stansfield, A. and Page, Ian, "The Design of a New FPGA Architecture," *Proceedings of the Forth International Workshop on Field Programmable Logic,* Oxford, England, September, 1995.

51 Rupp, C., "CLAyFun Reference Manual," National Semiconductor Corporation, Santa Clara, California, July, 1995.

RAM-based or serial design becomes a bottleneck for the system, being limited by the amount of time needed to access the various configurable resources in the system as well as the controller's ability to simultaneously schedule and program separate jobs effectively. As the size of the system grows, the demands on a global controller become increasingly greater until it is the limiting factor in performance. Further, the communication latency inherent in a large system limits the speed with which configuration data can be distributed using a RAM-based approach.

3.3.2 Wormhole Run-Time Reconfiguration Description

Wormhole Run-Time Reconfiguration (RTR) attempts to address the weaknesses of FPGAs when used in a computational environment, and to provide a framework for implementing large-scale rapid Run-Time Reconfigurable CCM platforms. It is intended as a method of rapidly creating and modifying custom computational pathways using a distributed control scheme (*data-driven* partial run-time reconfiguration). Similar to the data movement process in data flow machines, the onset or completion of computational streams are used to instigate the process of reconfiguration. However, Wormhole RTR is not limited to data flow computation and, as discussed below, can be adapted to work with other computing paradigms. The essence of the Wormhole RTR concept is formed from independent self-steering *streams* of programming information and operand data that interact within the architecture to perform the computational problem at hand. The method of computation itself can be compared to that of Pipenets (Section 3.1), in which multiple intersecting pipelines (streams) are formed within the processor to perform a task.

Wormhole RTR is based on the stream concept, which is an extension of the more common use of the term as defined in Section 2.6.2. In this case, a stream is a concatenation of a programming header and operand data. The programming header is used to configure a computational pathway through the system as well as to configure the operations to be performed by the various computational elements along the path. The stream is self-steering. As it propagates through the system, configuration information is stripped from the front of the programming header and used to program the unit at the head of the stream; thus, the size of the header diminishes as the stream propagates through the system. The stream header is composed of an arbitrary number of packets of programming information. Each packet contains all the information needed to configure a designated unit in the system. The composition and lengths of the packets are variable so that different packet types may coexist within the same stream header and hence heterogeneous unit types may be traversed by a given stream.
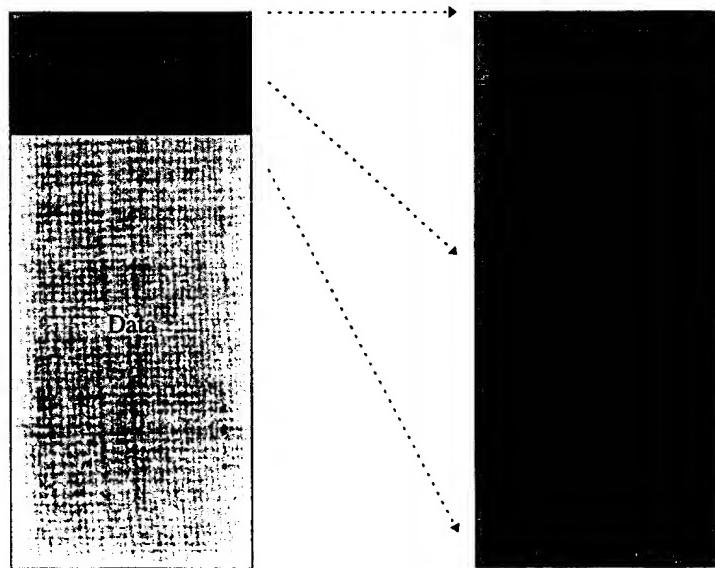
Figure 3.2 – Stream Format.

Figure 3.2 shows a stream that may be used with the Colt CCM. The Colt architecture was designed with Wormhole RTR in mind. The Colt prototype IC consists of six 16-bit bi-directional data ports, a 4×4 cylindrical mesh of Functional Units (FUs) and a 16-bit × 16-bit multiplier producing 32-bit results; all of which are connected through a "smart" crossbar network. Each of these units strips a packet from the front of an arriving stream header and stores configuration information from it. As shown, the stream header contains packets to program the various units of the computing platform in the order that they would be encountered along a path. In this case, the first packet is used to configure a data port, and then a crossbar packet that directs the stream to a particular column of the mesh. Once in the mesh, four separate packets configure Functional Units to perform independent functions on the data section of the stream. Finally, a second crossbar packet directs the stream out another data port, which is configured with the last packet in the stream header. Following the stream header is the data to be processed along that path through the Colt.
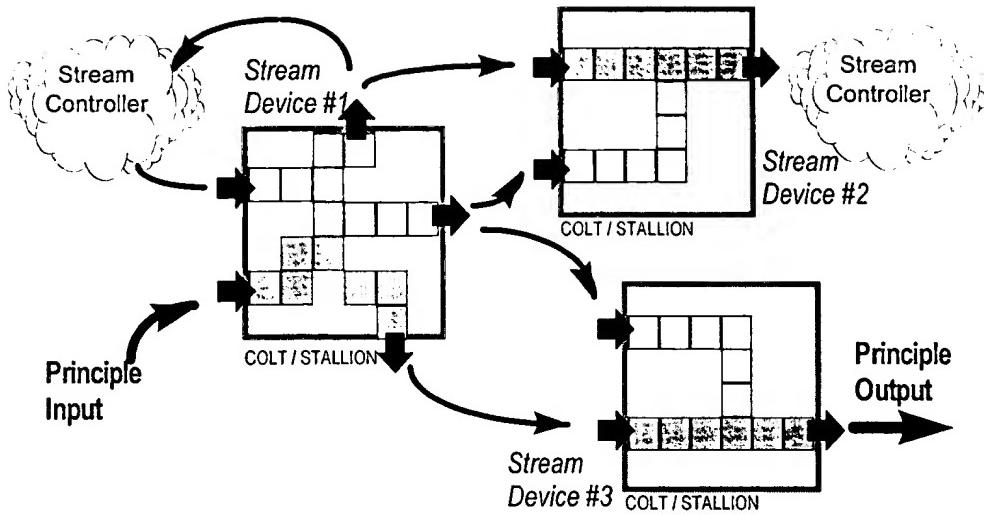
Figure 3.3 - Generalized Colt/Stallion Wormhole RTR Stream Processing Concept.

Note that the stream can be of arbitrary length. Thus, the path made through the system

by the stream header can be arbitrarily long, allowing deep pipelines to be easily formed across

multiple devices. Likewise, the data section of the stream can also be arbitrarily long (such as

could be found with the stream of values emanating from an A/D converter), allowing a single

configuration to process multiple operand sets without reconfiguration. This ability again can be

likened to the fetch/execute cycle of a microprocessor where the ratio of fetches to executions is

generally 1:1. However, in a system using stream processing an 1:X ratio is achieved where X is

the number of operand pairs that may be processed without reconfiguration. This represents a

significant savings in overhead that would normally be associated with fetching opcodes.

Further, a savings in power is realized because the state of the configuration information need not

change between operand pairs.

The exact path taken by the stream through the system can be determined at run-time,

allowing several competing processes to allocate resources as they become available. The

80

process of resource allocation is not unlike that found in operating systems employing Banker's Algorithm [52] to allocate I/O resources. A similar method could be used in this situation. However, in order to maintain maximum flexibility, this was not directly implemented on the Colt. Instead, this function will be implemented externally by Stream Controllers that arbitrate for resources both on and off-chip and substitute parameters for the physical resources allocated at run-time into the stream header. The process is similar to a normal operating system loading an executable and substituting for relocatable addresses.

The data section of the stream is sent through the pipeline of processing elements. There are several options for the formatting of the operands within the data section. One option is to send operands through the pipeline sequentially. In some circumstances, it may be advantageous to run sets of operands in parallel through the data stream in order to save routing resources, etc. On the other hand, if the data rate is low or I/O ports are scarce, operand pairs may be doubled up one behind the other within the stream. Once the path and operations through the system have been established by the stream header, any amount of data (practically infinite) can be processed along that path. A continuous stream of data is preferable, though not required, to maintain a high pipeline utilization rate.

---

52 Dijkstra, E. W., "Cooperating Sequential Processes," *Technical Report EWD-123*, 1965, Reproduced in Genuys, F., eds., *Programming Languages*, Academic Press, pp. 43-112, 1968.

3.3.3 System Perspective
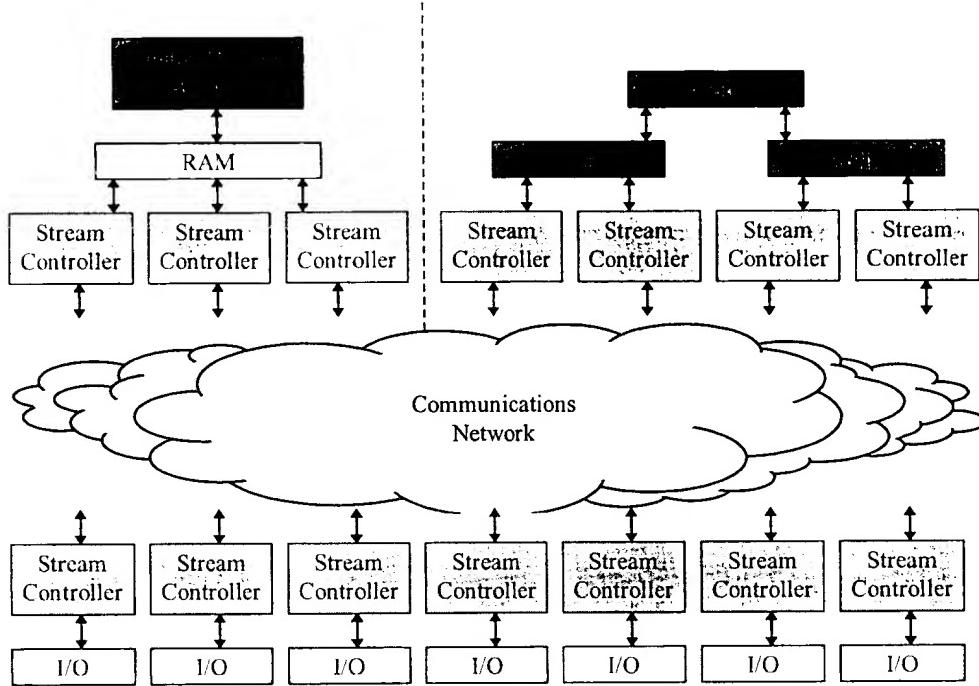
Figure 3.4 - Wormhole RTR System Perspective.

Figure 3.4 shows a higher level system perspective of one way to implement Wormhole RTR. In ths figure, the exact details of the communications network are left intentionally vague since it is not pertinent to the concept of Wormhole RTR itself. Along the top of the figure are two different types of processing boards that have been attached to the system. One of them is a board containing an Analog Devices SHARC chip and an attached RAM. On the other side of the dotted line is a board containing several Colt chips, as will be described in Chapter 4. The Colt chip is designed to work natively with Wormhole RTR, resulting in a simpler interface to the system. In fact, one of the Colt chips is shown as being attached to nothing but other Colt chips. A stream can flow to this chip from one of the others, reconfigure some of its resources

and then continue off of the chip. Since the SHARC is not designed with streams in mind, more interface hardware is required to connect it to the system. The program that it executes can easily be sent as a packet in the stream header. The program and the data that it operates on can be buffered into the RAM by the Stream Control units.

The Stream Control units are responsible for buffering the data from the communications network, preprocessing stream headers and scheduling stream injection into the system. The buffering process consists of the collection of data from the system and queuing it to remove the "gaps" in the data section of the stream caused by a bursty or slow data source, etc. Further, there is the problem of synchronization of data transmission. Through the communications network, the data can have arbitrary latency; however, at the interface to some chips, such as an FPGA or a Colt chip, synchronization and alignment of the data streams must take place to ensure that the operands are matched up properly at the various intersecting execution paths in the chip. Alignment isn't difficult since the Stream Controller can simply buffer valid data, which is assumed to be in the correct order. The problem of synchronization requires that all Stream Controllers involved in implementing a given page on the chip start and stop sending valid data into the chip on exactly the same clock cycle. Fortunately, this only needs to occur at the level of a given board, such as the board of Colt chips shown on the right.

The other functions of the Stream Controllers are more complex. In an effort to adhere to a distributed control philosophy, the Stream Controllers are given the task of scheduling and directing stream movement through the system. This is the process of determining available resources, altering the steering information in a stream header so that it is directed to those resources and then injecting the stream into the system, all while other Stream Controllers are

83

attempting to perform the same task with other streams. Of course, streams are not entirely independent entities and if any two streams have to interact, such as when computing the dot product of two vectors, the Stream Controllers must agree on a location in the hardware for the operation to take place. Further, there are more global problems of ensuring that deadlock does not occur in the resource allocation process. A competitive allocation scheme similar to that used in operating systems, such as Banker's Algorithm [52], could be used to resolve these issues.

3.3.4 Advantages & Disadvantages

In this section, the various merits and drawbacks of the Wormhole RTR approach are considered. The topics covered will be:

- Function Evaluation vs. Opcode Fetch Ratio

- Support For Heterogeneous Computing Environments

- Dynamic Implementation Sizing

- Object Oriented Streams

- Distributed Control

- Self-Timed Streams

- "Localized" Communications

- Fault Tolerance

- Design Complexity

- Stream Controller Design

- Data Flow Computing Paradigm

3.3.4.1 Function Evaluation vs. Opcode Fetch Ratio

Once a stream path has been established through the system, there is no need to reconfigure it again until the resources are needed for other computations. As stated, the stream can be infinitely long, allowing for 100% utilization of the processing elements along the path without need for the Von Neumann fetch-execute cycle for each operation performed. If CCM type elements are used in the system, the configuration information for them would only need to be sent in the stream header once. After that point, the CCM could process operands without the need for reprogramming for the duration of the stream. For this type of processing element, an X:1 instruction execution to opcode fetch efficiency is gained over control flow type implementations, which, though pipelined, have only a 1:1 efficiency in this regard. A cache can ease this burden; however, implementation of the cache costs hardware resources and, inevitably, cache misses will degrade performance. The X:1 gain results in processing efficiency increases both in terms of pure speed of computation and also in terms of the power dissipated, since the state of the control circuitry need not change.

Even if a control driven processor is included in the path of the stream, the program for the DSP could be sent in the header of the stream just as it would be for any other unit. This program would only need to be sent through the communications network once for the entire duration of the stream. Because of the localized nature of communications in Wormhole RTR, no further code would need to be transmitted, unlike, for example, a shared memory system in

85

which different processors must page in sections of code through the communications network on a regular basis throughout the execution of an algorithm.

3.3.4.2 Support For Heterogeneous Computing Environments

The flexibility of the packet format within the stream header allows the Wormhole RTR concept to be applied to a much broader range of computing than simply CCMs. The information carried within a packet can vary from several dozen bits needed to program a computational element within an FPGA to an entire program to be executed on a standard microprocessor. Thus, the types of units along the path traversed by the stream through the system may be very diverse. The stream is a conduit for control information and data that can independently guide itself and perform a computational task in a heterogeneous computing environment. The localization of the operations to be performed to data items contained within the stream itself, or between several interacting streams, simplifies not only the burden of communications overhead, but also eases the task of integrating normally estranged technologies to achieve optimal computational performance. In this sense, Wormhole RTR could also be applied to a Macro Data Flow system as presented by Gaudiot, et al. [53], in which a data flow graph topology is mapped onto a computational system consisting of a heterogeneous set of computational nodes of varying granularity.

The Colt CCM itself is an amalgamation of different computational resources. Essentially, the basic elements of the Colt are standalone units that may be connected in a variety

of ways by a stream. The crossbar is used to facilitate these connections and to maximize the flexibility given to the programmer as to the exact data flow graph implemented. There are no inherent interdependencies between the design of the different units of the Colt. Using Colt's rich interconnection resources to connect the explicit computational dependencies of the data flow graph, the Colt programmer may create an optimally parallel implementation of the algorithm. This is, of course, subject to the availability of resources, but in a full scale system resource restraints need not play a factor while, at the same time, all the inherent advantages of the Wormhole RTR methodology will still apply.

3.3.4.3 Dynamic Implementation Sizing

The localized nature of the communications involved with Wormhole RTR allow different sections of the data flow graph being implemented to expand or contract dynamically at run-time to utilize as much hardware as the current algorithm and existing system constraints allow. As different processes compete for hardware resources and the size of the physical machine increases or decreases, different portions of a given data flow graph could be implemented directly in hardware. For instance, if more hardware were added to a given system it could be detected by the Stream Controllers and used to configure more of the data flow graph directly in hardware; realizing an instant speedup with no modification or recompilation of the data flow "program" itself. As another example, the amount of physical resources in the system may remain the same, but a new process may be introduced. Existing processes on the machine

---

53 Gaudiot, J. L., and Ercegovac, M. D., "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proceedings of the 4th International Conference on*

could dynamically shrink the amount of code that is directly implemented in hardware to allow room for the new process to execute in a reasonable amount of time.

Stream Controllers can preprocess the stream header in much the same way that a program object file is preprocessed in a standard microprocessor. The program object file contains the assembly code to implement the program and many relocatable data and code addresses. These addresses are designated as relocatable because in general the compiler has no knowledge of where the program will eventually be loaded into memory at the time the program object file is written. When the program is loaded into the computer the operating system is responsible for replacing the relocatable addresses with the physical memory addresses that have been used for the particular execution of the program. For the purposes of a stream, the header itself can be stored as a kind of object code in the Stream Controller. When the stream is to be "run" or injected into the system, the Stream Controller can poll the system to find out what resources are available that the time. The Stream Controller can then configure the stream header so that the stream will to use whatever resources are available.

3.3.4.4 Object Oriented Streams
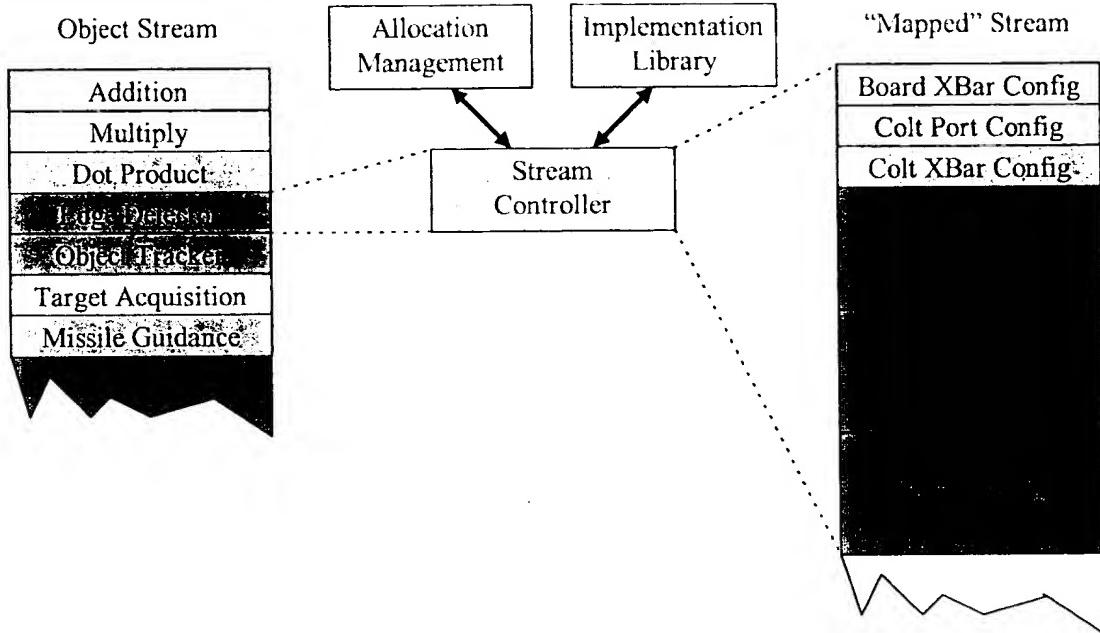
Figure 3.5 - Object Oriented Stream Concept.

Figure 3.5 shows the extension of the concept of dynamic process expansion and shrinkage to allow the possibility of object oriented streams in which the stream header simply contains "opcodes," each of which indicates the type of operation to be performed without necessarily dictating the exact piece of hardware to be used for implementation. A Stream Controller could receive such a stream, compare the opcode against a library of possible physical implementation strategies and then implement the most efficient method possible given available hardware resources by substituting the appropriate implementation into the stream header in place of the opcode.

In this way, a Wormhole RTR system could also implement the concept of generalized hardware in which the exact hardware to be utilized at run-time need not be known at compile

time. This applies not only to hardware resource availability, such as would be used for the dynamic expansion and shrinkage discussed, but it also opens the possibility of running a program on future hardware that has yet to be added to the system, or has yet to even be invented, without having to recompile or re-code. All that would be required is the addition of the implementation library for the new hardware to the system; dynamic stream to hardware mapping by the Stream Controllers could then immediately utilize it at run-time.

For instance, if a multiplication operation is required there may be several different units capable of performing it. There may be a dedicated multiplication chip, a SHARC chip and a Colt chip for use in the system. However, at the time the dedicated multiplier may be flagged as being in use. The Stream Controller then chooses to either substitute programming data for the appropriate device into the stream header. For the SHARC, this would be a short program. For the Colt chip, this could consist of either directly steering the stream into an on-chip multiplier, or the multiplier could be built from other chip resources. The Stream Controller would choose an implementation method from the available hardware resources, insert the appropriate stream steering and configuration code and then inject the stream into the system. Note that the selection of implementation method would have to be a cooperative decision between the set of all Stream Controllers that have intersecting streams in the system. If a dot product is being performed, for example, the pair of Stream Controllers that contain the two vectors must agree on the specifics of which type of resource to use and then which one. Using this type of dynamic resource allocation at run-time on a stream by stream basis makes it possible for a given application to grow and shrink with the system, not only in the static sense of the addition and removal of hardware, but also in coping with the dynamic problems of run-time system load.

Negotiation and allocation of resources could take place over a global control bus that connected all the Stream Controllers in the system. Using this bus, the Stream Controllers could each keep track of what resources are currently avalaible and which streams are currently being processed. Using this information, each Stream Controller could make intelligent decisions about the best method for implementing the stream(s) that are currently pending within it. At the same time, other Stream Controllers would be making the same types of decisions. The global control bus would be the only common point of communication in the system. The traffic over this bus should be small next to the amount of data being processed for sufficiently long streams.

The global control bus could also serve as the means for querying the implementation library shown in Figure 3.5. A Library Module could be placed on this bus, from which the Stream Controllers could obtain implementation strategies for any given "opcode" that was required. The strategies would contain resource requirement summaries and other information to allow the Stream Controller to make an intelligent choice between them given the current state of the system. Once the choice of implementation strategies had been made, the configuration header fragment needed to implement that strategy could be downloaded from the Library Module to the Stream Controller over the global bus. This could then be inserted into the final stream header after minor address relocations, etc. The Stream Controllers could cache implementation strategies and configuration header fragments, in addition to the current system resource allocation tables, inorder to mininimize traffic over the global bus.

The object oriented stream concept is important for experimental platforms. At present there is a large code base developed for the SHARC chip to implement the prototype communications system for which this hardware platform is targeted. It would be advantageous

91

to be able to plug a SHARC into the computing platform and use that existing code base with little modification. Obviously, this advantage applies to any upgrade path. The only standardization required between components is a port standardization and the stream concept.

Naturally, not all of the diverse devices that could be plugged into the system inherently support the stream concept. In those cases, such as for the SHARC, the processing element itself can be put on a supporting daughter board that conforms to the stream concept and interacts with the rest of the system. The proper environment for the SHARC can then be created on the board so that it can execute control driven code on data that has been stored in a normal RAM. The program it executed would be delivered to the daughter board as a packet in the stream header and the data would come from the data section of the stream. When processing is completed the data can be reformatted for the stream and injected back into the system.

3.3.4.5 Distributed Control

Because the streams independently guide themselves through the system using the information contained in the stream header, the configuration process is inherently distributed. Multiple independent streams can wind their way through the Colt chip simultaneously. The streams can all originate from independent and distinct external controllers, or streams can be initiated by the onset or completion of other streams in the system through the Stream Controllers. Likewise, part of the system can be used to process data operands while any set of neighboring units can be accepting configuration data from the header of one or more other streams, thus allowing overlap of the configuration/execute cycle and of process multitasking

within the same system. These possibilities are all offered due to the distributed nature of Wormhole RTR.

3.3.4.6 Self-Timed Streams

Another facet of Stream Controller design is the ability to buffer the data and handle data alignment and synchronization. This means that the latency through the communication network can be variable. This fact makes it possible to use an asynchronous clocking scheme. There does not need to be any concept of a global system clock at all. Data can be self-timed as it is sent through the communications network with an extra communications line attached to act as an independent clock. Transitions of this clocking bit could indicate new data items to be latched from the stream. Such a scheme would greatly simplify the timing problems that plague large board design with high speed system clocks. The Stream Controller would buffer up data sent through such a channel and then forward it to a particular processing element on the local board using an independent local clock. Synchronization between Stream Controllers that are being utilized for the same page on the local board could handle differences in data flow rates and latencies through the self timed communications channel.

3.3.4.7 "Localized" Communications

The distributed control scheme offered by Wormhole RTR also provides advantages for system scalability. Communication operations during both phases (configuration and execution) of stream processing are localized in nature in the sense that the stream flows from one unit to only a few others. Long latencies between distant units within the system can be effectively hidden through natural pipelining, thus allowing the overall speed of the system to increase over

that of a centrally controlled machine. Since the source and destination of the stream can be separated by arbitrary distances, the data channels themselves can be moderated by synchronization mechanisms that tolerate long latencies and are normally used only in Wide Area Networks (WANs); such as TCP/IP. Thus, the system could be truly distributed, across the room or across the world, accessing distant farms of configurable computing resources as they become available to create computation systems of truly global proportions. All the while the processing elements themselves could maintain high utilization rates.

Distributed control also has the advantage of promoting lower overhead in communications. Though the total amount of communications overhead in terms of time spent in the system may be higher, the percentage of time spent on overhead at the local level (defined as that seen by a given processing element), is reduced. This is due to the fact that there are generally fewer devices in the immediate area of a processing element and it needs only resolve contention and other conflicts with the local set rather than being forced to contend with all the elements in the system.

3.3.4.8 Fault Tolerance

Finally, as touched on above, a distributed control scheme better lends itself to fault tolerance. As long as a global controller is present in the system, it not only represents a fundamental limitation on computation on speed, but it also represents a prime target for a single point failure. In a well designed distributed system, any given element can fail and the rest of the circuit can continue to function by routing around the defect.

3.3.4.9 Design Complexity

Wormhole RTR encourages distributed control. Each resource, be it processing or routing element, can be designed to have a very "localized" view to control. It need only be designed to interact with the immediate processing area, as defined by the computing elements that immediately surround it. This leads to many control circuits, one for every element in the system. Individually, these are less complex than a single global controller would be; however, in total they may consume more area in silicon. The advantage is that little global control is needed, meaning less area devoted to routing and decoding signals. As the trends of VLSI tend toward more useable area in silicon it becomes increasingly difficult and area inefficient to route long control signals. A general rule of thumb for VLSI design practice recommends local complexity over global routing [54].

A good example of global control can be found in the Xilinx XC6200, which can be programmed similarly to a RAM. Implementing this requires decoding the target address and sending the decoded signals down long signals that must propagate across large areas of the chip in short amounts of time. Not only is it difficult to drive these signals in a reasonable amount of time, but the driving circuitry also requires power. Further, the long wires and extra drivers consume space that could be used for other purposes.

Designing Colt has shown that it is generally easier to design components for global control than it is to design for distributed control. Though the individual control circuits seem

---

54 Seitz, Charles L., "Concurrent VLSI Architectures," *VLSI Algorithms and Architectures: Fundamentals*, N. Ranganathan, eds., IEEE Computer Society Press, Los Alamitos, California, 1993, pg. 16.

simpler at first glance, creating a system of units that work in harmony with one another can be difficult. A discussion of how these types of problems arise and how they were solved will be included in Section A.3 on the detailed design of a Wormhole RTR crossbar.

3.3.4.10 Stream Controller Design

As discussed above in Section 3.3.3, the full design and function of the Stream Controllers is a complicated process involving problems on several levels. Of course, it is possible to simplify the problem by using a less distributed approach. Though it would cut into the possible parallelism, a more static allocation philosophy could be used to simplify the problem. For a system that implements two mutually exclusive tasks, a fully static implementation could be used. In such a design, one configuration of the system would be used during normal operation and later a second configuration could be injected via a second set of streams to perform another operation. Thus, the Stream Controllers would simply store two sets of streams and could be given a signal when the other set needed to be injected so that the system could be reconfigured to perform the next function.

One feature of streams for hardware implementation is the sequential access pattern to memory dictated by the queuing model. Under this assumption, the Stream Controller can make use of interleaved memory architectures to meet whatever memory access requirements are needed. In fact, the memory access time can be totally hidden by the interleave strategy, leaving nothing but the computation itself to limit system speed. Further, there is no need for caching since the address of the next memory access is entirely deterministic. Inevitably, there will be a need for some type of random access to the stream data. This problem could arise in the

calculation of an FFT, for example. For these situations, the system could generate a stream consisting of a set of RAM addresses. The Stream Controller would receive this stream and dereference the addresses in it and replaced them with operands that were retrieved from a local RAM. In a sense, the Stream Controller would then be acting as a cache for the system. This type of scatter/gather functionality would be simple to build into the Stream Controller and it would aid the implementation of some applications immensely.

3.3.4.11 Data Flow Computing Paradigm

A disadvantage of using streams for processing is the implicit use of sequential data access. By definition, the data in a stream goes through the machine in a pipelined fashion and the system depends on the fact that the stream is not reordered. It is possible, however, to use the Stream Controllers to reorder the stream if necessary to perform a given computation, or even to buffer the stream data at a processing element (containing a SHARC chip for example), perform a random access computation, and then send the stream data on to the next processing element. In this way, the stream concept can be considered as a method of moving data through the system rather than as a constraint that hinders performance. Removing sections of the stream in this way will lower system performance to that of the random access processing element, but that element can be as simple or as complex as needed to meet the performance goals of the system.

Some research has been done into the area of clustering operations under a larger data flow paradigm in this way. Gaudiot, et al. [53], have described the advantages and disadvantages of such a macro data flow system. Discussed are the tradeoffs of lost parallelism due to clustering large amounts of a data flow graph into one executable unit, including loss of fine

grain parallelism. Also discussed are several advantages, such as better performance when there is little parallelism in the algorithm. An analytical method for gauging performance is developed and simulation results are given for varying numbers of processing elements applied to the evaluation of a binary adder tree. As expected, the results show a dramatic speed up at first as processing elements are added, and then a gradual loss of speed as communications overhead dominates the computation when large numbers of processors are used to solve the problem. Hence, the incorporation of sequential processing elements into a stream based system could have advantages not only for random access data patterns but also for system performance overall.

The system will attain overall best performance when it is possible to act in fully pipelined mode, with no random access units delaying the stream. A data flow computing paradigm is most advantageous to satisfying this goal. Data flow computing specifies the program in terms of basic computational dependencies that specify the minimum set of restrictions on program execution order. Further, data flow programs naturally flow from functional programming languages in which data elements are computed and then are never altered. Program execution advances by creating new data elements from the existing set. This means that many programming structures, such as pointers, that encourage random data access are denied to the programmer, making it necessary to solve the problem using other methods. The process of thinking in these terms can be difficult to learn.

Also, data moves through the system from processing unit to processing unit just as operands move through the data flow graph from operator node to operator node. The translation process from a data flow graph to a stream based computer is greatly simplified over that of

converting standard imperative code, such as C, to a data flow or stream based architecture. One effort to perform such a conversion is [55]. Using this system, it is possible to translate C code to a data flow graph and then to map the data flow graph to a given target architecture. The resulting hardware realization is easily derived, though is not optimal in terms of what a human programmer could accomplish with data flow in mind.

3.3.5 Relationship To Message Passing

An interesting communications mechanism that is in some ways related to Wormhole RTR is a concept known as Active Messages [56]. Active Messages are a means for implementing a synchronization mechanism between processors in a multiprocessor network. The authors cite several machines in which a three phase handshake takes place between the producer and consumer processors to ensure proper transmission. This generates a great deal of communications overhead as the two processors devote time to the operation.

Rather than effectively polling for data by going through the handshake process, Active Messages provides an interrupt driven system of I/O for exchanging messages. The code for handling the data contained in a given message is assumed to reside on the destination processor at a known address. When the source processor transmits the message containing the data, the

55 Peterson, James B., O'Connor, R. Brendan and Athanas, Peter M., "Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 17-19, 1996.
56 Eicken, Thorsten von, Culler, David E., Goldstein, Seth Copen and Schauser, Klaus Erik, "Active Messages: a Mechanism for Integrated Communication and Computation,"

address of the receiving process on the destination processor is included in the header of the message. The arrival of the message at the destination processor triggers an interrupt of that processor, which then runs the routine at the address indicated by the message. This routine quickly receives the data of the message and schedules the handling process to run. By directly placing the message contents into the data space of the destination process, overhead is reduced by removing the need for operating system layers to buffer, sequence and transfer the data to the destination process.

Further, the authors point out that much of the overhead of conventional communications schemes arises because of the need for processing and communications to be mutually exclusive tasks; the processor must poll for data during the communications phase. The interrupt driven system offered by Active Messages allows the communications and processing phases to overlap. This also has ramifications for the communications network itself. In other systems, the communications network must be designed to be very fast in order to minimize the time during which the processor is "busy waiting." However, if communications and processing can be overlapped, then the communications network needs be no faster than the speed at which data is processed. This same argument can be applied to the success of Wormhole RTR in that communication delay is effectively hidden by the inherently pipelined nature of communications and processing.

This system is in some ways similar to Wormhole RTR because the address of the receiving process is essentially akin to the configuration information stored in the stream header.

Proceedings of the 19th International Symposium on Computer Architecture, Gold Coast,

In fact, when incorporating processors into a Wormhole RTR system that do not support the concept natively, the Active Message mechanism could be used to trigger processing of the data stream. The first stream header that reaches such a processor may contain the entire program needed to process the data section of the stream. Succeeding stream headers could then simply refer to the starting address of the previously downloaded code. Even in a CCM device the Active Message concept may be used to trigger the use of various "sets" of configurations, or alternatively, to trigger Stream Controllers to inject numbered configurations into the CCM device.

3.3.6 Data Dependent Computation

One of the inherent problems of using a pipeline for computation arises when it is necessary to evaluate a recurrence relation as in:

$$X(i) = f(a(i), X(i-1), ..., X(i-M))$$

This posses a problem for a pipelined processor because the depth of the pipeline dictates that the computation of result $X(i)$ cannot be completed until result $X(i-1)$ has been completed. Thus, it would seem that each element must be sent through the pipeline singly before the computation of the next element may begin. When the basic function can be assumed to have the form:

$$X(i) = f(a(i), X(i-1))$$

Kogge has published a solution to this problem where a companion function $g()$ is found that can weaken the dependence on prior terms to:

Australia, May, 1992.

$$X(i) = f(g(a(i), a(i-1)), X(i-2))$$

This then can be used to form two smaller problems:

$$X(2i) = f(g(a(2i), a(2i - 1)), X(2(i - 2)))$$

$$X(2i + 1) = f(g(a(2i + 1), a(2i)), X(2(i - 2) + 1))$$

The problem can then be further divided into three problems of size n/3 or four problems of size n/4. Kogge points out that a cost tradeoff is reached when the complexity of the subfunctions grows much greater than the complexity of the orginal compuation [57]. What is important is that the dependence of the *ith* element can be pushed back to elements computed much earlier in the recurrence relation so that the needed results can propagate through the entire pipeline before they are needed for the computation. Ideally, the recurrence relation should be reduced to the point where none of the previous K elements are needed to complete computation of the next value, where K is the latency of the pipeline.

Kogge goes on to discuss the optimization of looping structures in a pipelined machine. There is no formal method for optimization as he described for recurence relations, but he does indicate that increasing the complexity of each loop iteration can lead to better pipeline utilization and, hence, speedup. He gives several examples including three methods for implementing maxtrix-vector products and methods of implementing Fast Fourier Transforms (FFTs) [58].

57 Kogge, Peter M., *The Architecture Of Pipelined Computers*, New York, NY, Hemisphere Publishing Corporation, 1981, pp. 181-186.

58 Kogge, Peter M., *The Architecture Of Pipelined Computers*, New York, NY, Hemisphere Publishing Corporation, 1981, pp. 186-192.

This work can be directly applied to Wormhole RTR where deep computational pipelines are formed. In the presence of deep pipelines, the ith computation may require the result of computation $i-1$. Because of pipelining, result $i-1$ may not be available at the time needed to perform computation i. This method of decomposition may provide a way through which the need for result $i-1$ may be delayed until it is in fact available.

4. Colt Architectural Overview

The structural and physical design of the Colt CCM is a major thrust of this work. The details of the implementation, to a great extent, reflect the applications for which it is intended. However, many of the features, including native support for Wormhole RTR, are useful in a more general sense for the design of CCM devices, and for more general computing. This chapter presents the larger issues behind the design of the Colt CCM and a brief discussion of its features.

4.1 Factors Affecting Speedup

Some desirable characteristics of CCM devices can be gleaned from a comparison with conventional microprocessors. A CCM's speedup gain over conventional technology is of particular importance.

$$T_{CPU} = \sum_{i=1}^{N_{ITER}} \frac{T_{SETUP\_CPU}(i) + T_{EXEC\_CPU}(i)}{N_{UNITS\_CPU}} \qquad (4.1)$$

$$T_{CCM} = T_{SETUP\_CCM} + \frac{\sum_{i=1}^{N_{ITER}} T_{EXEC\_CCM}(i)}{N_{UNITS\_CCM}} \qquad (4.2)$$

The equations above define the total time spent in executing a given iterative task for a conventional microprocessor (T_{CPU}) and for a CCM device (T_{CCM}), respectively. They model the behavior of the core computation of computationally intense tasks, such as simulations, signal

104

processing and ... CCM devices provide the most benefit for tasks in which large amounts of data are subject to the same algorithmic steps; thus, the parameter N_{ITER} is included in the equations. It refers to the number of data items that will be processed using a given page or algorithmic step once it is configured onto the device. This could also be thought of as the number of iterations through a loop as the elements of an array are processed, for example. The value of some parameters varies with the iteration number, as indicated by the index i. Average values over all iterations will be used for these parameters, so that the equations simplify to:

$$T_{CPU} = N_{ITER} \bullet \frac{T_{SETUP\_CPU} + T_{EXEC\_CPU}}{N_{UNITS\_CPU}} \qquad (4.3)$$

$$T_{CCM} = T_{SETUP\_CCM} + \frac{N_{ITER} \bullet T_{EXEC\_CCM}}{N_{UNITS\_CCM}} \qquad (4.4)$$

T_{EXEC_CPU} and T_{EXEC_CCM} refer to the time actually spent processing data. This is the total delay through the fixed function units in a microprocessor, or the delay through the configurable units of a CCM. For purposes of comparison, the assumption is made that the T_{EXEC} parameters are both equal to 1 clock cycle for the CCM and the CPU implementations. That assumption is made on the grounds that a CCM can always be made to contain execution units that can operate at the same rate as those of a microprocessor. In fact, a CCM could even be considered as an amalgamation of special purpose units, all connected in a data flow manner using Wormhole RTR. The selection of the types, sizes and flexibility of the units that are included on a CCM should be driven by the demands of the types of applications that will be executed on it. Since the locality of communications inherent to Wormhole RTR provides the means of moving operands between the units of a CCM at rates comparable to those found in microprocessors, it

seems reasonable to assume that the unit-for-unit performance of the CCM can be equivalent to that of a microprocessor.

For the CCM device, T_{SETUP_CCM} refers to the amount of time required to configure the CCM to perform a given task. This is a function of the configuration I/O bandwidth of the device and the number of configuration bits required. For the CPU, T_{SETUP_CPU} encompasses the total time spent performing opcode and operand fetches, recovering from branch prediction errors, pipeline stalls due to data dependencies, move instructions, etc. For purposes of discussion here, T_{SETUP_CCM} will be computed assuming that the multiplier plus all 16 FUs of the Colt must be programmed, plus 6 data ports and 10 crossbar connections for an average of ($16*8 + 6 + 10$)/17 = 8.47 clock pulses per computational unit. T_{SETUP_CPU} is a more difficult number to quantify. From Flynn [59], the number of cycles per instruction (CPI) for a pipelined microprocessor ranges from 1.25 to 1.6, where the execution time of basic instructions is assumed to be a single clock cycle and the fractional component is due to the effect of branches, operand inter-dependencies, instructions with longer execution times and ALU loading delays. The high end of the range is for processors performing floating point operations (which tend to have long execution times) and the low end is for processors performing purely integer operations. These numbers can be equalized to assume that all instructions require one clock cycle to execute by subtracting out the fractional increase due to run-on (long execution time) instructions. These components add 0.37 CPI for the floating point number and 0.09 CPI for the integer processor number, giving a range of 1.16 to 1.23 CPI. Finally, it should be noted that a

Wormhole RTR processor exploits a native register forwarding ability in that no explicit move-type instructions need be executed. To distinguish this difference from CPU architectures, it should be recognized that Flynn [60] cites that 54% of the executed instructions in scientific computing using a Load/Store (RISC) type architecture are moves. Thus, to normalize the CPI range for the microprocessor, an additional overhead can be computed as $0.54/(1-0.54) = 1.17$ move instructions per non-move instruction execution. The overhead of executing these instructions must then be added into the range as $1.16*(1+1.17) = 2.52$ and $1.23*(1+1.17) = 2.67$, giving a final range of 2.52 to 2.67 CPI for the CPU. Subtracting the normalized execution time of one clock cycle from this ranges gives a range of 1.52 to 1.67 for the T_{SETUP_CPU} parameter. In fairness, because the Colt is designed with integer based processing in mind, the integer processor value of 1.52 CPI will be used for the T_{SETUP_CPU} parameter in all further calculations.

Finally, N_{UNITS_CPU} and N_{UNITS_CCM} give the number of simultaneously executing units on each device. For the CCM, this would include all the functional units that are participating in the computation. For a CPU, this number is generally significantly smaller. There are many reasons for this, not the least of which is that the CPU contains a great deal of overhead logic to support the Von Neumann computing paradigm while maintaining acceptable utilization of the computing resources. Also, a CPU is limited by the inherent parallelism in the sequential program being executed. A data flow processor such as Colt can fit many more execution units because this silicon overhead is not needed; nor does the data flow graph specification impede

---

59 Flynn, Michael J., *Computer Architecture: Pipelined and Parallel Processor Design*, Boston, MA, Jones and Bartlett Publishers, Inc., 1995, pg. 258-259.

the exploitation of concurrency. For a superscalar architecture, Acosta [61] explains that for an architecture containing two integer adders, two floating point adders and two floating point multipliers, all of which are pipelined, the asymptotic speedup is approximately 1.75. The speedup reaches steady state at this point primarily because of inter-operand dependencies in the calculations. This will be considered to be an upper bound on N_{UNITS_CPU}, since, though more units could be added, their contribution is minimal.

It could be argued that this limitation should apply to the CCM device as well. However, these inter-operand dependencies are explicitly described by the data flow graph and the dependencies between them can be avoided by properly scheduling the configuration of the various pages onto the architecture. Thus, maximum parallelism can be exploited at all times, without the need for the processor to wait for intermediate results.

The main advantages of CCM devices over CPU devices is evident in Equations 4.1 - 4.4. In a CCM device, the T_{SETUP_CCM} time is incurred only once, whereas this overhead is encountered for every iteration (or execution) of the code in a microprocessor type device. Also, a CCM may utilize many more I/O pins to transfer data to the computing elements. A CPU must devote a large percentage of pins to addressing and other "house keeping" chores. In a real-time system that is well suited to a data flow representation, the concept of addressing may not be required for the majority of processing. Such would be the case if the output of an A/D converter

60 Flynn, Michael J., *Computer Architecture: Pipelined and Parallel Processor Design*, Boston, MA, Jones and Bartlett Publishers, Inc., 1995, pg. 146.
61 Acosta, R. D., Kjelstrup, J. and Torng, H. C., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions on Computers*, C-35:815-828, September, 1986.

were directly connected as an input to a CCM processor, for example. Further, a CPU must

devote large areas of silicon to other support units, such as cache and address generation. This

silicon can be used for computational units on a CCM. The difference in execution time between

the two implementations will then primarily be a function of the relative numbers of executions

units, the difference in setup times and the number of iterations being performed. The overall

speedup of a CCM implementation over a CPU implementation is:

$$S = \frac{T_{CPU}}{T_{CCM}} = \frac{N_{UNITS\_CCM}}{N_{UNITS\_CPU}} \bullet \frac{N_{ITER} \bullet (T_{SETUP\_CPU} + T_{EXEC\_CPU})}{N_{UNITS\_CCM} \bullet T_{SETUP\_CCM} + N_{ITER} \bullet T_{EXEC\_CCM}} \qquad (4.5)$$

This equation was used to generate the plots below.
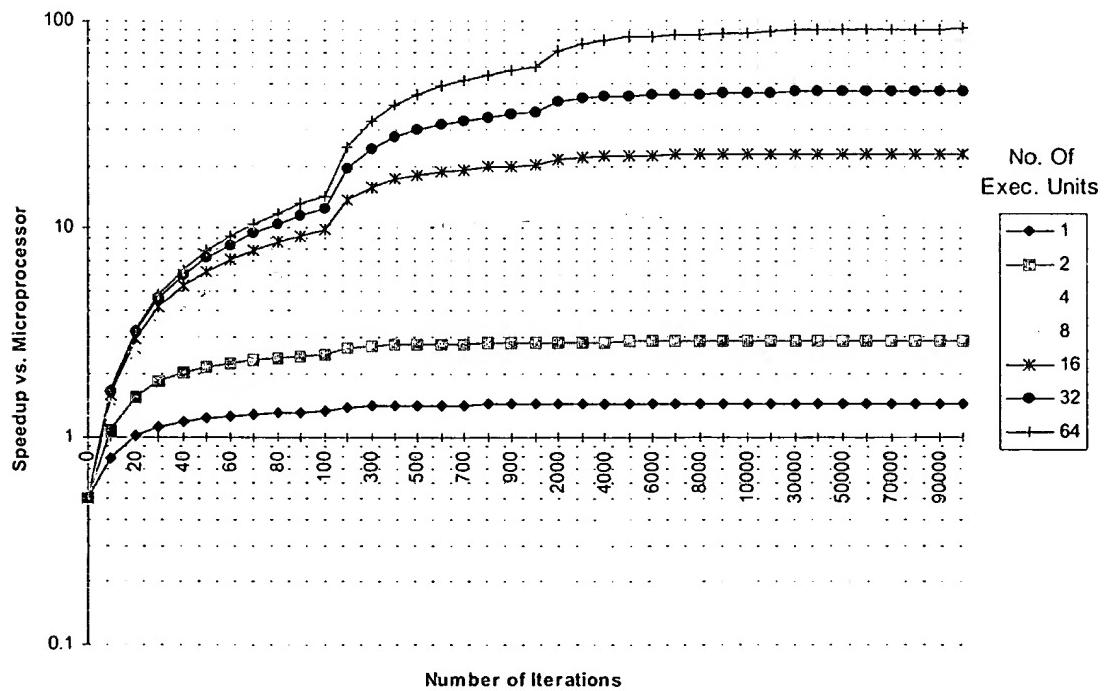
Figure 4.1 - CCM vs. CPU Speedup For Various Numbers Of CCM Execution Units.

Figure 4.1 shows a family of curves generated using speedup Equation 4.5. The values assumed are: $T_{EXEC\_CPU} = 1$ CPI, and $T_{EXEC\_CCM} = 1$, $T_{SETUP\_CCM} = 8.47$, $T_{SETUP\_CPU} = 1.52$ CPI, $N_{UNITS\_CPU} = 1.75$ and N_{UNITS_CCM} varies from curve to curve. The CPU setup time greatly affects the asymptotic speedup value. The execution times for both the CCM and the CPU are assumed to be 1; thus, the asymptotic value of the second term of the speedup equation is $(1+1.52)/(1) = 2.52$. This is then multiplied by the ratio of execution units to obtain the overall asymptotic speedup. Curves are plotted for various numbers of concurrent CCM execution units, from 1 to 64 CCM execution units. The number of iterations, or executions, of the code is used as the x-axis. Notice that as many as 20 iterations are required before the CCM reaches the break even point. This is a direct result of the CCM requiring a longer initial setup time. However, the CCM setup time is a onetime cost, and over more iterations it becomes negligible.
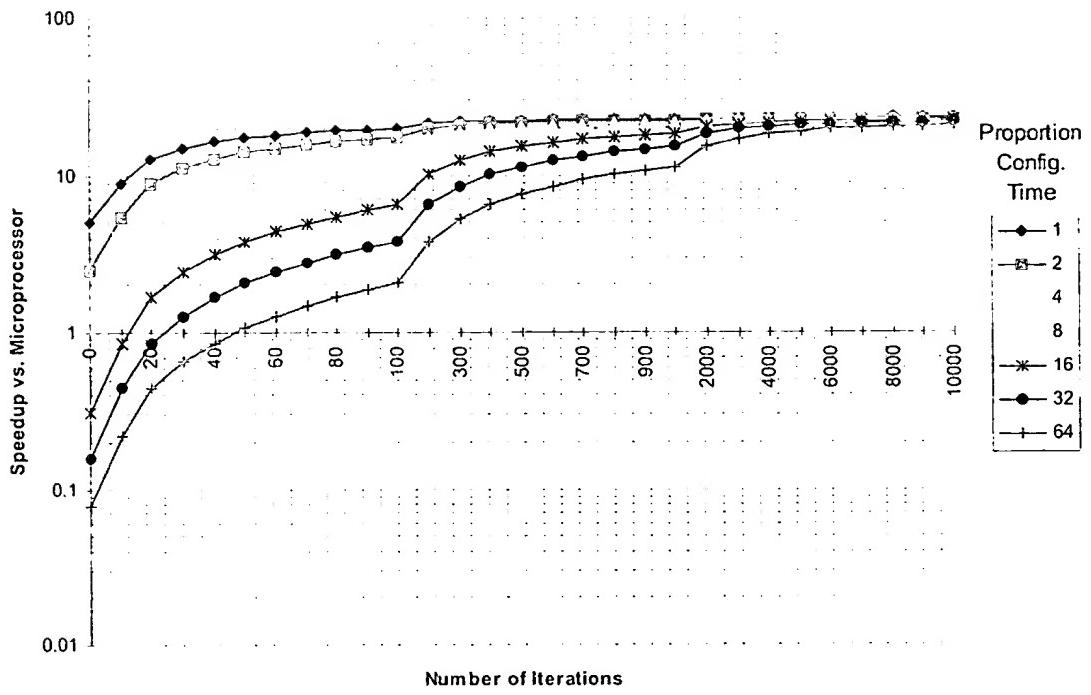
Figure 4.2 - CCM vs. CPU Relative Setup Time Effects.

Figure 4.2 shows the affect of various CCM setup times on the actual speedups achieved.
This plot was made assuming the same values for the parameters, except that the number of
CCM execution units is fixed at 16; resembling Colt, and the parameter varying between curves
is the CCM setup time. As the setup time of the CCM approaches that of the CPU, the
importance of large numbers of iterations becomes less and less. This is shown by the top curve
where the CCM setup time is 1. However, as more and more setup time is required by the CCM,
the ratio grows and the number of iterations becomes very important to the speedup that is
gained. The curve for the Colt device, having a setup time of 8.47, is similar to the curve
representing a setup time of 8 cycles.

The graphs bring about two conclusions. First, for small numbers of execution units, a given configuration of a CCM must be used for multiple operand sets before it becomes useful as an acceleration device. Thus, the number of data items that are sent through the pipeline created by the Wormhole RTR stream should be maximized in order to realize the greatest speedup. Second, the number of configuration bits required to setup a CCM device should be kept low if the device will be used in an environment where it is difficult to supply long streams of operand data. This is equivalent to the case where the device will need to be reconfigured many times in quick succession to produce a final result.

An introduction to the developmental background of CCMs can be found in a paper by Acosta, et al. [62]. The architectures of most contemporary CCMs, and the systems based upon them, have been implemented with traditional FPGA roles in mind. Such features as single bit computational cells, global control structures and restrictive routing resources that are sufficient for providing glue logic prove prohibitive to efficient data flow implementation. Further, the amount of overhead required to configure a contemporary FPGA is made excessive by this bit level approach, which adversely impacts not only the density of computing elements in the circuit silicon, but also the reconfiguration time of these chips. Since faster reconfiguration time can translate directly into faster computation, the relatively slow reconfiguration times of many FPGAs is a great hindrance to competitive computing speeds. Also, the speed of operation for designs mapped to most of these devices is slow compared to ASIC solutions, making

---

62 Acosta, E., Bove, Jr., V., Watlington, J. and Yu, R., "Reconfigurable Processor for a Data Flow Video Processing System," *Field Programmable Gate Arrays (FPGAs) for Fast*

comparisons less than favorable. Worse yet, the maximum attainable clock rate varies greatly with the design being implemented. Such variations can be disastrous for a real-time system and can prove troublesome for any rapidly reconfigurable system that is designed to maintain a processing schedule (such as a deeply pipelined system).

4.2 Data Flow Implementation

Large scale implementation of tagged-token dynamic data flow machines is expensive in terms of hardware and communications overhead [63]. For many problems, the computational advantages of a data flow architecture can be achieved using a (locally) static direct communication graph. This is the approach taken in the Colt/Stallion architecture. By directly connecting functional units and guaranteeing that only one operand exists on an arc at a given time, the implemented data flow graph is reduced to a set of intersecting pipelines. In this regard, the architecture resembles a Pipenet as discussed by Hwang, et al. [64]. Hwang describes the types of algorithms and looping structures that can be executed and the theoretical speedups that are achievable [65]. Colt extends these capabilities by providing special provisions for

Board Development and Reconfigurable Computing, SPIE - The International Society for Optical Engineering, Bellingham, Washington, pp. 83-91, 1995.

63 Traub, Kenneth R., Beckerle, Michael J., Papadopoulos, Gregory M. and Hicks, James E., "Overview of the Monsoon Project," *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Los Alamitos, California, pp. 150-155, October 14-16, 1991.

64 Hwang, K., *Advanced Computer Architecture*, New York, NY, McGraw-Hill, 1993, pp. 442-446.

65 Hwang, K. and Xu, Z., "Multipipeline Networking for Compound Vector Processing," *IEEE Transactions on Computers*, pp. 33-47, 1988.

conditional execution, as well as an innovative pipeline configuration scheme in the form of Wormhole RTR.

Recall from Chapter 3 that the hardware for such a system should consist of functional units with fairly diverse interconnection abilities to allow subsections, or *pages*, of the data flow graph to be directly mapped onto the system and to facilitate rapid swapping of pages with little overhead to the active computation. Each page is a piece of the overall data flow graph that can fit within the limitations of currently available hardware resources. Operands arriving on arcs entering a particular page are queued by the system until the programming process is completed. Once the page has been configured onto the architecture, the queues feed operands into the pipelines, producing new results on arcs leaving the page. These are either fed directly into the next page, if it has already been configured, or are queued until part of the system has been configured with the next page. Thus, pages can be dynamically programmed into the system at run-time, paging in and out as necessary much as an operating system pages sections of programs.

4.3 Word vs. Bit-Oriented Approach

The Colt/Stallion architecture is targeted at DSP-type processing. As such, most computation will be performed on word-wide operands as opposed to bit-oriented operations. To take advantage of this, the Colt has been designed with chainable 16-bit data paths and processing units throughout the chip. Whereas other FPGA type processors use a single bit data path, the bus wide grouping of bits in the data path of the Colt requires less control circuitry and

switching to steer operands around the chip. The advantages of this are three fold. First, less silicon area is needed for routing than in a bit oriented design (routing is extremely resource-intensive in silicon). Second, less power is consumed for circuitry used to route the signals. Third, there is a higher predictability of propagation times for signals, making it possible to better pipeline designs. Following the same philosophy, the functional units on the Colt process operands which have sizes consisting of multiples of 16-bits; performing the same operation on all bits in a word. Since the functional units are larger, and are capable of a 16-bit computation every clock cycle, fewer units are needed than would be required in a bit-oriented design. Thus, higher speed operation can be achieved because fewer switches must be traversed to proceed from computation to computation. Further, the locality of the bits within a word allows for the construction of faster structures for computations that require high speed communication between neighboring bits. Addition, subtraction and shifts all exhibit this requirement.

Perhaps the greatest gains in using a word oriented approach lie in the number of configuration bits required. Since the same operation is performed over the entire 16-bit word, only 1/16 the number of configuration bits are required to specify the operation compared to a bit oriented architecture. Again, the gains from this are many fold. Fewer storage elements are required to store configuration bits; saving silicon area, power and routing resources necessary to program. Further, because fewer bits are necessary to specify a complete configuration, fewer bits need to be moved on and off the chip when performing a reconfiguration and hence the overall reconfiguration time can be lower. The inherent disadvantage, of course, is that applications requiring fewer than 16 bits of precision will suffer from a kind of internal fragmentation of resources since the least common denominator of computational grain size is

115

fixed. This waste was considered secondary to the advantages to be had, especially in light of the particular computational tasks that Colt is targeted towards; tasks that will benefit from 16 bits of precision. Colt is one embodiment of a set of higher concepts; an implementation that was to some extent dictated by real-world goals.

The net result of going to a word-oriented approach over a bit-oriented approach is a gain in computational density over traditional bit-oriented FPGAs. The gain in density is partly spatial, owing to the fact that fewer switches, storage elements and control logic circuits are required. The gain is also temporal in that fewer configuration bits are required; allowing faster reconfiguration times. Finally, because fewer switching elements need to be traversed to perform a computation, the raw clock speed of the Colt can be higher, boosting the computing power even further.

4.4 Partial Reconfiguration

As discussed in 3.3.1, it is desirable to have the capability of independently programming different parts of a CCM. Just as in the larger overall system, the Colt chip is designed to allow one part of the chip to be executing an algorithm while another part is being reconfigured to do the next computation. This ability allows more of the complicated interconnection network to be placed on the chip and less of it at the board level, reducing overall system cost and maintenance problems. If a system were constructed from many contemporary FPGAs, the designer is forced into a one page per chip allocation policy, thus more chips would be required to exploit the parallelism between pages. In many contemporary FPGAs, the entire chip has one program; it

must be reconfigured with an all or nothing philosophy. This has the effect of wasting chip resources for pages that are I/O bound and don't require all the computational resources that exist on the die. Whereas, a chip that has independently programmable sections can execute the I/O intensive page while the rest of the chip is set up for the next page. The process is very much like a pipelined instruction cycle in a microprocessor. While the processor is executing the current instruction it is already fetching the next one. Being able to partially reconfigure a chip allows pipelining at the CCM programming level.

4.5 Arithmetic Computations

Another impetus for making a custom chip is the need for a chip that is geared specifically towards arithmetic computations. Many problems, such as FIR filtering, can be thought of as a large data flow graph operating on words of data. The smallest unit of computation in the Colt chip operates on pairs of 16-bit words rather than on pairs of bits as is the case for FPGAs. A limited amount of special purpose hardware has been included on the Colt to deal specifically with numeric computation. This includes a fixed point multiplier as well as hardware in each Functional Unit (the FU is the basic computational structure in the Colt) to deal with issues of floating point and fast integer multiplication. Normalization, barrel shifting, and multiplication/division have received particular attention.

4.6 Colt Architecture
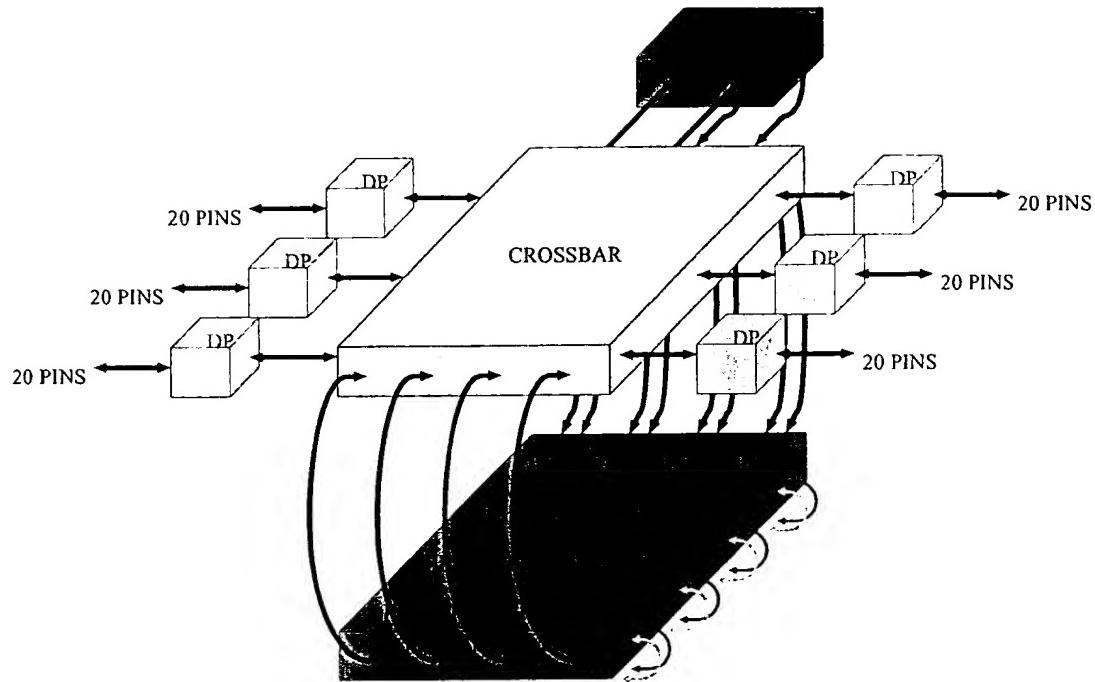
Figure 4.3 - Colt Architecture Overview.

An overview of the Colt chip architecture is shown in Figure 4.3. Much of the philosophy of the Colt organization has been explained above. The chip supports simultaneous programming/execution, fast reconfiguration and word level processing rather than bit level. The Stream Controllers allocate and then program a path through the chip through one of the data ports. The stream proceeds through the crossbar and then to either the mesh of Functional Units (FUs) or to the multiplier. The stream can then go back through the crossbar to any part of the chip or off chip via a data port.

118

4.6.1 Data Ports

These move data on and off of the chip and there are six of them on the Colt. They consist of a bi-directional 16-bit data path, and four bi-directional control pins named Program, Write, Transmit and Receive, for a total of 20 pins per data port. Note that no address pins are included. This is because the concept of an address isn't used in the data flow graph itself. Addresses may be used by the Stream Controllers to store data streams in temporary RAM, but that is only a convenience to simplify the RAM hardware. The Colt chip does not use them; at least in the same sense as they would be used in a Von Neumann machine. At one time, addresses were used in the design for use with lookup tables and other such control flow programming structures; however, the concept was dropped to save pins. In order to maintain high utilization of chip resources, high data throughput must be maintained. One bottleneck to that goal is the number of pins dedicated to moving data.

As it turns out, pins are the scarcest resource when designing in VLSI today. The reasoning for this is made readily apparent by an application of Rent's Rule [66], which is expressed mathematically as:

$$P = K B^p$$

Where P is the number of pins that are required by the circuit, K is a technology dependent constant, B is the number of functional blocks in the system and p is another technology dependent constant. Simply put, Rent's Rule states the number of pins required by a circuit

---

66 Radke, C., "A justification of and an improvement on a useful rule for predicting circuit to pin ratios," *Proceedings of the 1969 Design Automation Conference*, pp. 257-267, 1969.

grows as a power of the number of functional blocks. In VLSI terms, one can well imagine a square die of width W, with area W^2 and perimeter $4W$. Since in most processes today pads can only be placed around the perimeter of the die, the designer is faced with the problem of useable die area (functional blocks) growing as W^2, but the number of available pads (pins) growing only as $4W$. Rent's Rule states that the number of pins required will also grow as W^2, giving rise to the fact that pins are fast becoming the scarcest resource in VLSI technology.

By not using addresses, about 24 pins are saved per data port since that many address pins would be needed to attain a reasonably sized address space of 16 Megawords. The absence of addressing pins doubles the number of possible I/O ports. It could be argued that address and data values could be multiplexed on the same pins in the style of an Intel 80x86 chip, but extra time would need to be added to the bus cycle to allow for these values to propagate out of the chip. Rather than effectively halving the speed of the bus cycle in this way, and essentially nullifying any gains that have been made, this idea was dropped as well.

Each data port is bi-directional and can be programmed to operate in three different modes: *Raw Mode, Synchronization Mode* and *Loop Mode*. Raw Mode allows the data section of the stream to flow into the chip as fast as possible. No effort is made to align or synchronize the data with other data ports that may be receiving streams that intersect with this one. In Synchronization Mode, limited flow control is implemented, guaranteeing that the data sections of different streams will be synchronized so that the right pairs of data items arrive at the right place at the right time. Loop Mode extends the data alignment features of Synchronization Mode to further guarantee that only one set of valid operands will exist on the chip at a given time. There are six data ports on the Colt chip and a given data port can be configured to synchronize

120

with any subset of them, regardless of whether the ports are reading or writing data. A data port can even be configured to synchronize with itself.

Obviously, since the ports are bi-directional they can be programmed to work as either input (read) ports or output (write) ports. With one notable exception, the three modes are identical when the data port is programmed to function in output mode. The first important point is that only valid data is written from the chip. The flow control pins of the port will signal the Stream Controller when invalid data is written so that it will not be recorded in memory. Also of significance is the fact that the data port will write out valid data whenever it arrives from inside the chip regardless of the readiness of the Stream Controller. Even if the Stream Controller signals that it is not ready to receive data, the data port will write out any valid data that arrives from the chip. A wait signal from the Stream Controller can be made to cause the input data ports to stop accepting new data; however, any data that has already entered the on-chip pipeline will be written out.

No further effort is made to stop the flow of valid data from being written from the port because it is impractical to do so. In order to stop the flow of valid data that has already been injected into the chip, the entire pipeline along the path through the chip would need to be stopped in a single clock cycle. It is not feasible to back propagate a halt signal along this path for several reasons, not the least of which is that it has almost arbitrary length. Further, the exact path through the chip is chosen at run-time and the number of steering switches involved in constructing the path could be quite large, adding even larger amounts of propagation delay.

4.6.1.1 Raw Mode
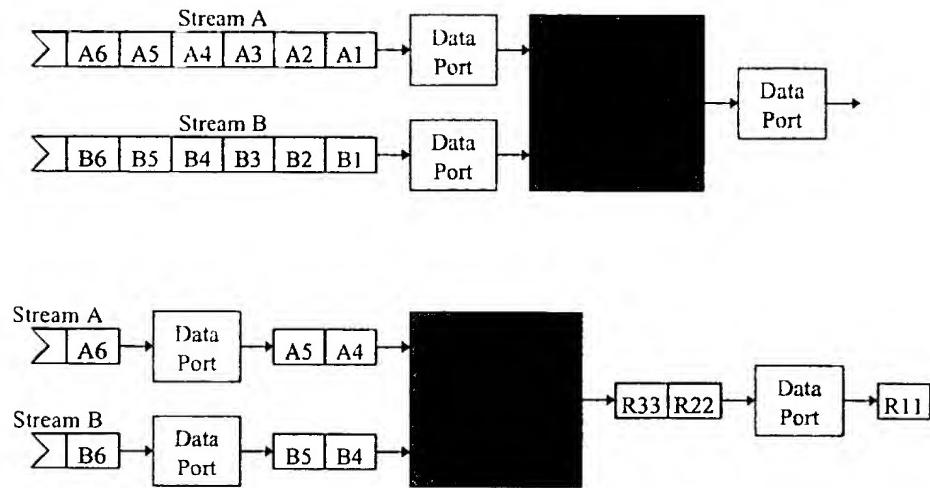
Figure 4.4 - Data Port Raw Mode Operation.

The general operation of Raw Mode is shown in Figure 4.4. The colored data items in the streams represent valid data and the clear items are invalid data. The invalid data may have been caused by a number of factors, including differing execution path lengths, data acquisition irregularities or the sporadic data production characteristics of conditional execution. The upper picture shows the two streams before they enter the chip and the lower picture shows the situation at some later time. The data items are combined through a calculation into a result stream. Each item in the result stream is labeled Rxy where x is the item number of Stream A and y is the item number from Stream B that was used to create the result.

Raw Mode forwards the two streams onto the chip unhindered. Notice that data items A4 and A5 are both allowed onto the chip even though they are both invalid. Further, valid item A5 is aligned with invalid item B5 so that an invalid result R55 will be calculated; effectively destroying item A5. Item A1 was also destroyed in this manner, which is indicated by invalid

122

result R11 exiting the data port. As previously mentioned, this result is flagged as being invalid by the data port so that the Stream Controller can treat it appropriately.

4.6.1.2 Synchronization Mode
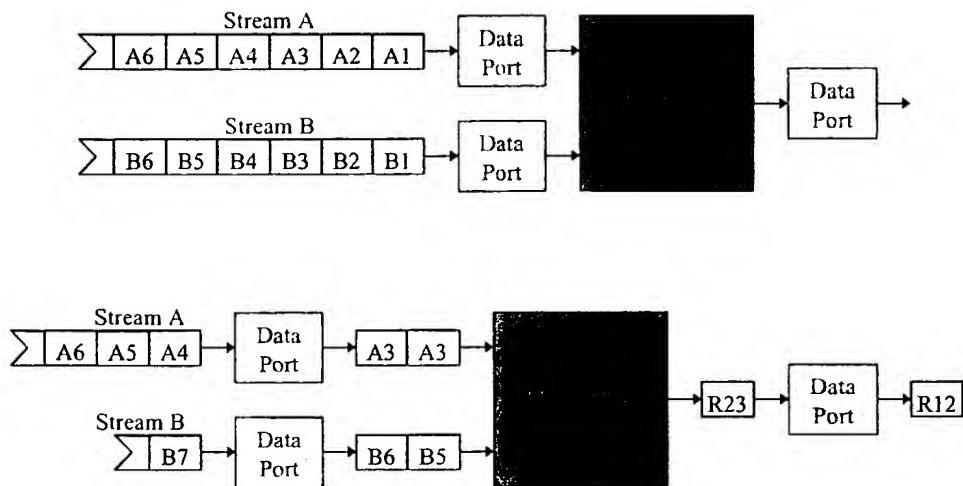
Figure 4.5 - Data Port Synchronization Mode Operation.

Synchronization Mode ensures that complete sets of valid data items will enter the chip simultaneously so that no valid data will be effectively destroyed as item A5 was in the Raw Mode example. Figure 4.5 shows Synchronization Mode working on the same two streams as were used in the Raw Mode example. Here, Stream A was held up until valid data from Stream B could be mated to data item A1, and again for data item A3 until item B6 arrived. The valid results R12 and R23 have been produced, reflecting the skew caused by the flow control actions of the data ports. At the time when invalid item B5 entered the data port, the A3 Stream Controller was signaled to halt and an invalid form of A3 was injected into the chip. When valid item B5 arrived at the data port, both streams were allowed to proceed and valid versions of A3 and B6 entered the chip simultaneously. R36 will be the next valid result produced.

123

4.6.1.3 Loop Mode
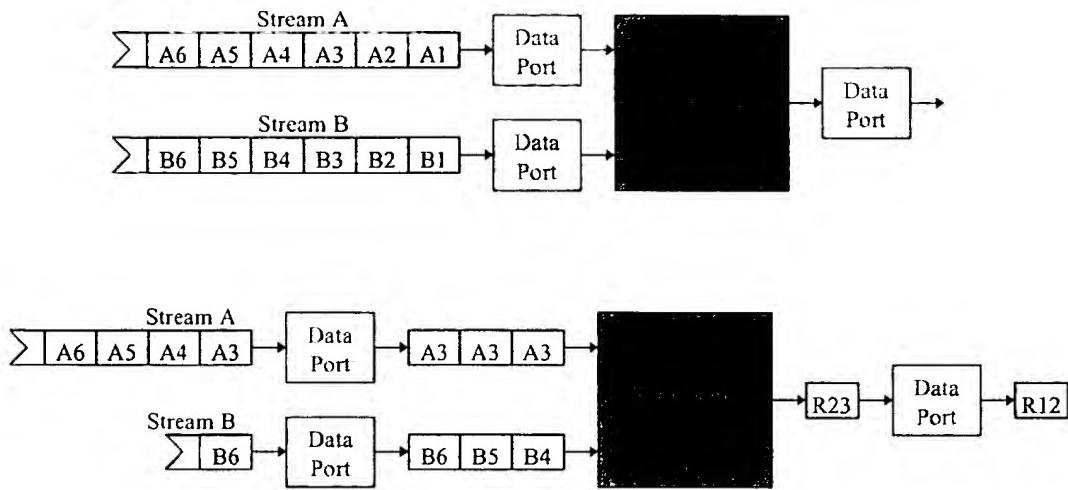
Figure 4.6 - Data Port Loop Mode Operation.

Loop Mode is an extension of Synchronization Mode. One use of Loop Mode is as flow control for conditional looping within the Colt chip. The concept is to first ensure that all data ports being used for the looping page either have data available (in the case of a read port), or are ready to accept data (in the case of a write port). Then, a single valid data item is accepted into the chip at each read port simultaneously. This is processed within the chip, possibly looping back on reentrant stream paths. When processing is complete, a valid result is forwarded to a writing data port and another valid data item is accepted at each of the read ports. The chip continues in this way in lock step fashion. As this process continues, the same data alignment procedures used for Synchronization Mode are enforced. Figure 4.6 shows the operation of a system in which the data ports are configured to used Loop Mode. The picture is similar to that of the Synchronization Mode example, with the exception that the A3 and B6 operands were not immediately accepted as a valid data pair. This is because the result R23 has not yet exited the

124

chip. When result R23 has left the chip, valid versions of A3 and B6 will be injected into the chip simultaneously.

4.6.1.4 External Programming

Programming information, in the form of a stream header, can enter or exit through a given data port independently of any of the other ports. When injecting a stream header into the chip, the data port functions as a read port in Raw Mode. The port will accept the stream unconditionally and this mode overrides all others. Invalid data that is latched during programming mode is ignored for configuration purposes. The data port itself strips off one 16-bit word of the stream header's configuration data and then forwards the rest internally to the rest of the chip. When the end of the stream header is reached, the port begins operating in either read or write mode as it has been configured.

4.6.1.5 Internal Programming

A stream header may also arrive at the port from inside the chip as flagged by an internal Program signal. This Program signal overrides any mode that the port may be in except external stream header configuration. In this case, the port acts as if it were in output mode. The first word of the stream header is latched and used as the programming information for the data port. After the first word of configuration information has been stripped from the stream header, the remainder of the stream is forwarded to the Stream Controller, which must immediately begin accepting the new stream. Invalid data in the stream header is flagged by the data port so that it will be ignored by the Stream Controller. When the end of the stream header is reached, the data

port reverts back to either read or write mode as it has been programmed. There is a one clock cycle latency going in either direction through a data port.

4.6.2 Crossbar Network
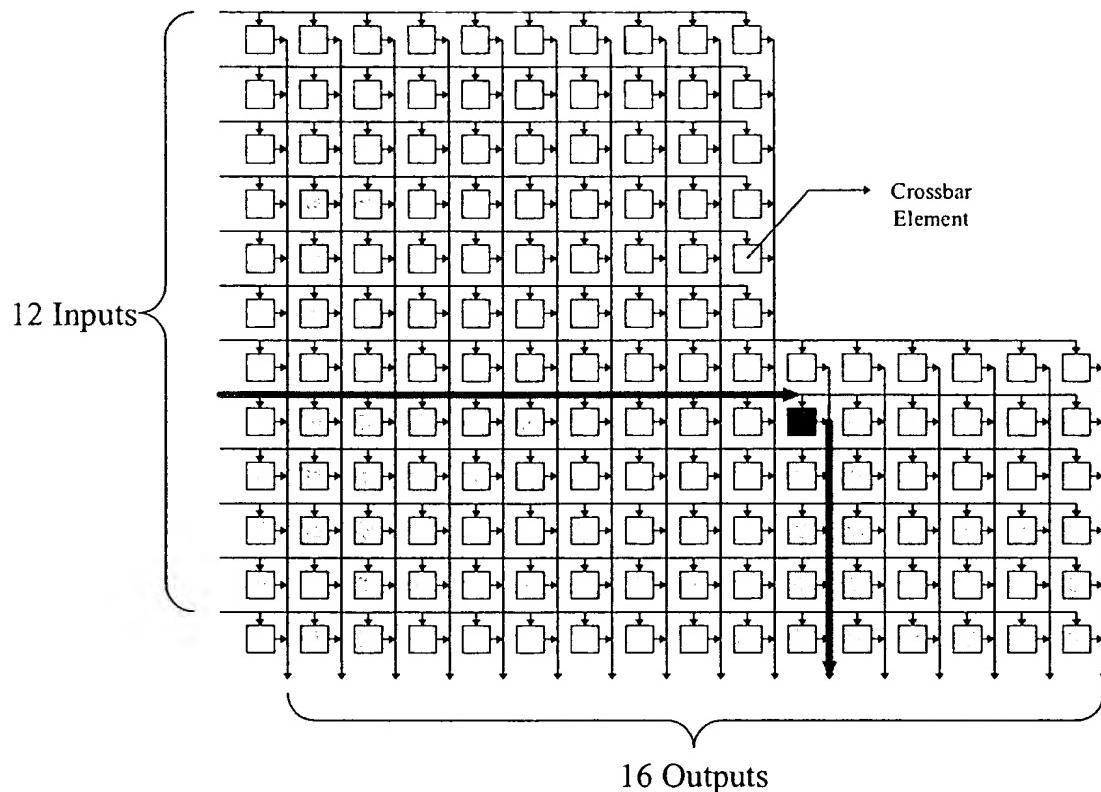
Figure 4.7 - Basic Crossbar Construction.

From a data port, the crossbar is the next unit that a stream will wind through. Unlike more standard networks that are switched using a global controller, this crossbar is designed to support the concept streams; thus, a feed forward control strategy is used. Because of the distributed control aspects of Wormhole RTR, the crossbar must be capable of being switched at multiple points simultaneously. To support this, the crossbar was constructed using 156 loosely

126

coupled state machines. The detailed design of this "smart" crossbar can be found in Section A.3.

Originally, a full crossbar interconnecting individual Functional Units (FUs) was envisioned. This would have allowed for fine grain resource allocation and complete routing flexibility. After some size calculations however, it became apparent that it would not be area efficient to make even a modest sized crossbar network of 50 addresses on the chip. For example, allowing for 10λ separation between wires, 18 wires per data bus and 50 data buses side by side on a chip using a 0.5 μm (λ=0.3 μm) process, the cross bar would need to be approximately 10*18*50*0.3μm = 2.7 mm on a side. To much of the chip area would have been devoted to routing and not enough to the FUs that would be performing the actual work. Naturally, this is a judgment call that must be weighed against the loss of routing flexibility. Realizing that the chip area consumed by a crossbar network varies as the square of the number of nodes, a compromise was reached resulting in the scheme shown in Figure 4.3. There are 12 inputs into the crossbar and 16 outputs; consuming less than one fourth of the area used by a 50 node cross bar. The rest of the connectivity is derived from the mesh as described in Section 4.6.3.

In an effort to save chip real estate, even some of the 12x16 crossbar connectivity has been sacrificed. It is not possible to go directly from an input data port to an output data port through the crossbar. This decision decreased the implementation size of the crossbar by removing the mapping hardware from the six data port crossbar inputs to the six data port crossbar outputs. Thus, the area savings is approximately (6*6)/(12*16) = 18.75%, which is

depicted by the missing switch points in Figure 4.7. This, coupled with the fact that it seems a ridiculous waste of valuable data ports to go in one port and directly out the other, made this simplification a sound decision.

Programming the crossbar is simple. Each crossbar output has a unique address that is taken from the first word in the stream header. This address specifies which crossbar output to use and the remainder of the stream is forwarded on to that address. One possible path is indicated by the heavy red line in Figure 4.7. There are crossbar inputs from each of the data ports and from the multiplier, as well as one from the bottom of each of the columns of the mesh. Crossbar outputs go to each data port and to the multiplier as shown. Two crossbar outputs go to the top of each column of the mesh. There is a one clock cycle latency from crossbar input to crossbar output.

One could question the need for a crossbar. Indeed, the crossbar was questioned many times. However, the more appropriate question is what is lost without a crossbar? It is not a simply a matter of whether or not it is possible to map an algorithm to the mesh using reduced connectivity. The question is how much run-time reconfiguration ability is lost. In a full fledged system, it is desirable to be able to dynamically program parts of the chip while other parts are computing. It is much like an operating system loading up programs and running them over a long period of time. Just as in an operating system, the memory (chip resources) will become fragmented. It will be part of the programmer's and compiler's job to minimize this fragmentation so that chip resources aren't hopelessly muddled into tiny, unusable, fragments. Nonetheless, different regions of the chip will be used to do different things at different times. These regions will shift at run-time based on the algorithm and on the data set, making it

128

impossible to predict what regions will be available in the future. Likewise, at the system level, streams will move from resource to resource in an unpredictable manner. Connectivity to the ports at the board level will most likely be limited due to the difficulty of implementing even a modestly sized crossbar on a board. Thus, the only way to access any given part of the chip is by giving the ports full connectivity into the mesh so that they can be used to program and supply data to any part of the mesh at any time. The crossbar, though even in VLSI it is silicon intensive to implement, is an all but essential element for constructing a smoothly running stream driven system.

4.6.3 Multiplier

Multiplication is a common operation in DSP-type applications. So much so that almost all DSP processors contain a dedicated Multiply-Accumulate unit (MAC), as was discussed in Section 2.1.2. Thus, it is a safe assumption that the Colt chip will be called upon to do a great deal of multiplication as well. Unfortunately, multiplication tends to be resource intensive to implement in FPGA-type architectures and even then it is relatively slow.

Table 4.1 - Comparison of 16-bit Unsigned FPGA Multiplier Implementations.

Device	Serial		Parallel	
	% Resources	Rate (MHz)	% Resources	Rate (MHz)
Altera 81188	7	2.14	51	3.66
NS CLAy	3	0.91	45	3.60
Xilinx 4010	8	1.90	60	3.80

Table 4.1 shows some of the results compiled by Petersen and Hutchings on the implementation of multipliers in FPGAs [67]. Bit-serial and parallel implementations are shown for several FPGAs, along with the percentage of the chip resources that are required to implement these structures and the rate at which results are produced.

One solution to this problem is offered by Magenheimer, et al. [68]. There, the authors describe an algorithm for implementing multiplication by any constant using a few shifts and additions. For example, multiplication by 5 can be computed using the formula $4*X + X$, which can be implemented by a shift left by 2 and an addition. These operations can be implemented with ease in FPGA-type architectures. This implementation of multiplication was one of the driving factors behind including a barrel shifter in the FU. Multiplication only by constants would seem to be an overly strict constraint, but it appears quite frequently in normal code. The authors cite that over 91% of the multiplication operations that occur in programs are by a constant value [69]. Thus, the shift-and-add structure can be included as part of the data flow graph for the program. Further, since the Colt is a run-time reconfigurable device with special provisions for quick changes to established paths of execution, it will be feasible in many cases to change the shift-and-add configuration on the fly. This would be particularly true when a large

---

67 Petersen, Russell J. and Hutchings, Brad L., "An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing," *5th International Workshop on Field Programmable Logic and Applications*, Oxford, England, pp. 293-302, August 1995.

68 Magenheimer, Daniel J., Peters, Liz, Pettis, Karl W. and Zuras, Dan, "Integer Multiplication and Division on the HP Precision Architecture," *IEEE Transactions On Computers*, Vol. 37, No. 8, pp. 980-990, August, 1988.

69 Neuhauser, C. J., "Instruction Stream Monitoring of the PDP-11," Standford University Department of Electrical Engineering Computer Systems Lab, *Technical Note 156*, May, 1979.

number of operands all needed to be multiplied by the same constant. The Stream Controller could even substitute in the appropriate configuration for the desired multiplier at run-time as part of the "object oriented" resource allocation policy.

Nonetheless, there are applications in which multiplication by a constant, even if the constant can be changed relatively quickly with respect to a long stream of operands, is simply insufficient. Applications that require a real multiplier include dot product operations such as may be found in an FIR filter. Here the multiplier and multiplicand can both change with every clock pulse. To handle these cases, it was decided that a dedicated multiplier should be included in the chip architecture. The pipelined multiplier included on the Colt chip can handle 16-bit by 16-bit operands and produce a 32-bit result with a latency of two clock cycles. For the Colt, the multiplier only performs unsigned multiplication, though eventually it would be advantageous to performed signed arithmetic as well as division. Still, a multiplier as implemented occupies approximately twice the chip area as a single FU. This is contrasted with the many FUs that would be required to realize a 16-bit by 16-bit multiplier using them as building blocks.

4.6.4 FU Mesh

Inputs From Cross Bar
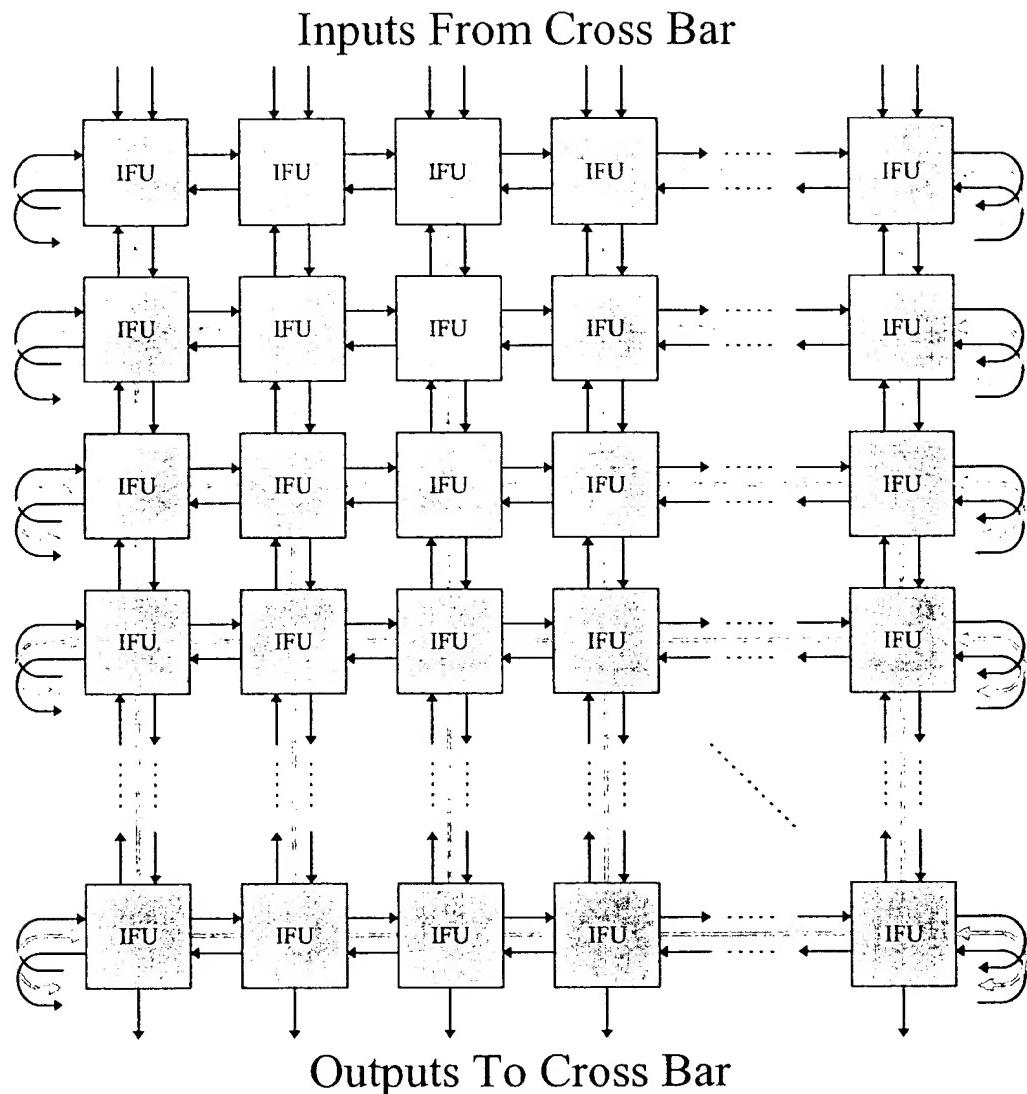
Figure 4.8 - Extended Mesh Topology.

One of the great compromises in the chip was the decision to embed the Functional Units

(FUs) into a mesh. This greatly reduces the interconnection possibilities between FUs versus the

132

available possibilities in a full crossbar; however, for large numbers of FUs the savings in chip area is substantial. Efforts have been made to enhance the possible connectivity between FUs in the mesh by adding extra connectivity resources to the standard topology. Though the dimensions of the modified mesh actually used on the Colt is 4x4, the concept of the topology is shown in Figure 4.8. The Interconnected Functional Units (IFUs) shown in the diagram are FUs that have network switching capabilities added to them. The IFUs are arranged in a standard mesh rectangle. At the top of each column of the mesh, two input feeds come directly off of the crossbar. One output back to the crossbar leaves the bottom of each column. Each IFU can communicate with its north, south, east or west neighbors via direct connections.

The enhancement to the standard mesh architecture is shown by the gray grid of buses superimposed over the IFUs. This represents the Skip Bus. The Skip Bus is a means of sending (skipping) operands over one or more IFUs in a single clock period. Remember the importance of timing the arrival of operands at a given node in the data flow graph. The Skip Bus is one way that need is satisfied. The Skip Bus can be programmed to steer operands from one IFU to another that is in a distant part of the cluster or it can be programmed to deliver an operand to an adjacent IFU. The switching capabilities added to an FU to make it an IFU are partly used to control the configuration of the Skip Bus. The Skip Bus is split into many short segments connecting adjacent IFUs, just as the hard wired adjacent neighbor connections. However, the Skip Bus can be programmed to pass a value from a segment entering an IFU to one or more other segment(s) leaving the IFU. Because the Skip Bus is broken into a mesh of segments, the pass through capability can be used to broadcast an operand to several units or to route many individual values between several different subgroups of units, even in the same row or column.

133

The flexibility that this provides adds greatly to the connectivity possibilities of the basic mesh without adding that much more routing area since the added buses are more or less local, just as the standard nearest neighbor connections. It does require added multiplexers to connect the different sections of the Skip Bus in various configurations, but these are rather small.

A stream header can pass through the mesh, configuring IFUs as it passes through. The stream can follow any path desired through the mesh topology. It can even come out the bottom of the mesh, go through the crossbar and come back in at another point. Further, the interconnection scheme better resembles a torus [70] than a mesh since the edge connections along the right and left sides are connected to their neighbors on the opposite edge. The only limitation to stream steering through the mesh is that only one stream can enter through the top of any given column. This is because one of the outputs from the crossbar connects to the northern local connection of the top IFU and the other crossbar output connects to the northern Skip Bus connection of the top IFU. Stream header data cannot be sent through the Skip Bus, only through local connections; hence, stream header data can only be sent through the crossbar output that connects to the local input of the top IFU. This was a design choice that was made to reduce the overall size of the layout of the FU, and possibly should be changed in future implementations. In Colt, operand data can be sent through the Skip Bus connection at the top of the mesh by terminating the stream header at the top of the mesh. An example of this is discussed in Chapter 5.

---

70 Hwang, Kai, *Advanced Computer Architecture*, New York, NY, McGraw-Hill, 1993, pp. 83-94.

4.7 FU Architecture
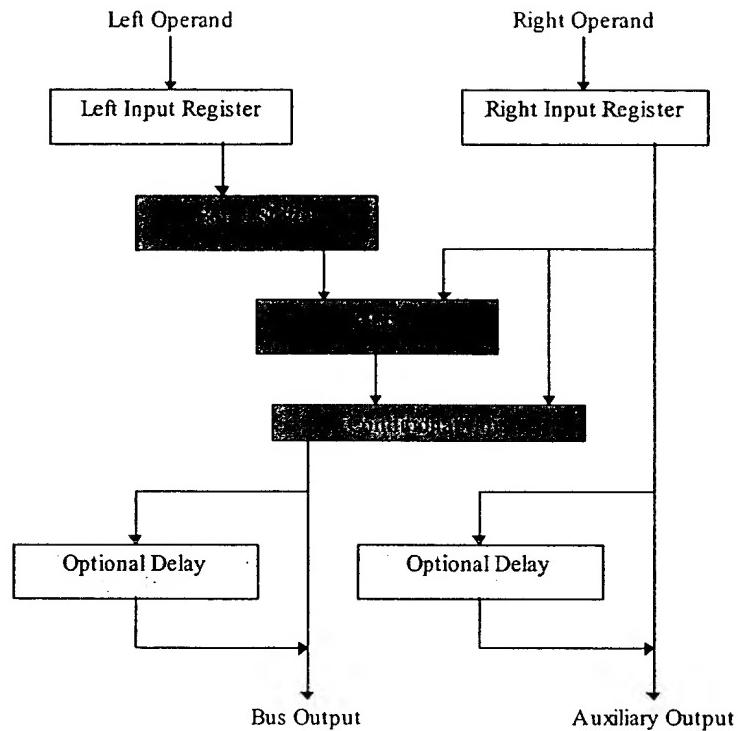
Figure 4.9 - Simplified FU Schematic.

The Functional Unit (FU) is the basic computational unit used in the system. A partial

schematic for it is shown in Figure 4.9. Two 16-bit operand inputs come into the unit through

the right and left operand registers. The left operand can then be conditionally passed through a

Barrel Shifter. The (possibly) shifted left operand and the right operand are then sent through an

ALU. The results of the ALU operation then pass through a Conditional Unit, a Barrel Shifter

and an optional Output Delay before leaving the FU.

135
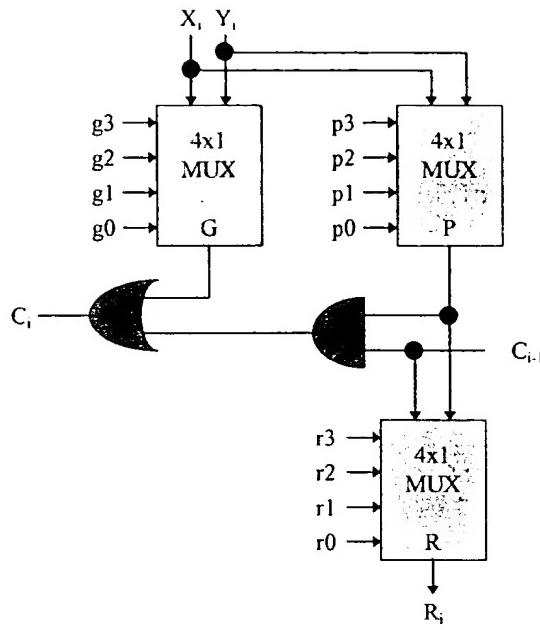
Figure 4.10 - ALU Bit Slice.

The function of the ALU is determined by the fully programmable bit sliced design shown in Figure 4.10, which is based on a similar design taken from Mead and Conway [71]. The three signals coming out of the multiplexers reflect the general carry look ahead principle of having Propagate, Generate and Result signals to produce the required arithmetic function. Using this configuration, the ALU can be programmed to perform addition, subtraction, incrementing, shift left, two's complement and many other operations. Further, it can be programmed to perform any of the sixteen Boolean functions of two variables.

71 Mead, Carver and Conway, Lynn, *Introduction to VLSI Systems*, Reading, MA, Addison-Wesley, 1980, pp. 150-154.

The Conditional Unit is used to allow the output of the ALU to be multiplexed with the right operand. The ability to do conditionally select between these is useful for conditional execution in general and for multiplication specifically.

The Barrel Shifter has been included as a basic FU component for two main reasons. First, it is nearly essential for efficient implementation of floating point arithmetic. It can be used for mantissa alignment when performing additions and it is also helpful for re-normalization of the mantissa after a floating point operation has been performed. The Barrel Shifter can be set to conditionally shift-right-by-1 or shift-left-by-1, -2, -3 or -4 bits. Second, the Barrel Shifter is quite useful for efficient implementation of integer multiplication by a constant as discussed in Section 4.6.3.

Finally, the optional Output Delay is used to insert a one clock cycle delay at the output of an FU in order to facilitate execution path equalization as is required for proper data flow operation without tags. There are no specific delay units on the chip, but rather this distributed delay approach has been taken. The main reason is that the introduction of another unit type into the system would greatly complicate allocation and routing problems on the chip. Whereas with this method, all units remain the same and there is always a delay nearby if needed.

4.8 Addressing

Each unit on the chip has a unique address. This address is stored in the lower six bits of the first word in every configuration packet in the stream header for the chip. As mentioned earlier, this address is not absolutely necessary for the stream concept. It does, however, allow

the addition of several enhancement features that make configuration more efficient. The stream concept would seem to lend itself to a relative addressing scheme in which the steering information in the stream header would merely indicate which direction to go at each cross road in the communication network. For instance, in the FU mesh, all the stream really needs to indicate is whether to go north, south, east or west. However, the programmer may wish the stream to split into two different paths mid-mesh. Then the relative direction approach would have to be augmented by some sort of control scheme to indicate which path a particular set of packets was destined for. If this were not added, then the relative north directive would proceed along both of the new paths, possibly leading to undesirable results. On the other hand, if an address is included in the stream packet, then there can be no doubt as to the destination of the packet.

Further, addressing allows the possibility of quickly sending a new stream, and stream header, along a preexisting path. When a unit receives a new stream header, but the first word in the first packet does not contain its address, the unit immediately forwards the packet in the direction that it was configured to do previously. Thus, a shortened stream header can be constructed that only alters a single unit along a reconstructed path, without having to alter any intervening units. This is a tremendous speedup advantage over the relative path finding approach. Of course, the relative approach could be augmented to perform this function as well, but it adds another level of complication. Also, in a situation like the crossbar, where there are sixteen different ways for a stream to go, effectively an address needs to be used, even in the "relative" approach. The relative addressing approach does have merit in that fewer encoded steering bits need to be included in the stream header. Also, the relative addressing approach

138

offers the ability to inject a stream into the chip from any point and have it map itself directly onto the architecture without having to have any absolute unit addresses changed to match the physical units being configured. Absolute addresses force the use of a relocation process for the various unit addresses, much as an operating system does with a relocatable program, before loading. However, these advantages seemed to be outweighed by the added overhead of the complexities of solving the other problems.

4.9 Broadcast Programming

The Colt chip also supports a type of broadcast programming. Binary 111111 is the reserved broadcast address. Only the address comparators for FUs and some of the crossbar outputs are designed to respond to the broadcast address. The data port comparators are not designed to respond to the broadcast address in order to save hardware. This function is pointless for the data ports. If a broadcast address were presented to a data port, the data port would be programmed just as it would have been if the more proper address had been used, nothing out of the ordinary would happen since there is no fan out at the entrance to a data port. Also, only the crossbar addresses that correspond to FU mesh inputs are designed to respond to the broadcast address. If the broadcast address is presented at the crossbar input, then the stream is routed through every output of the crossbar that connects to a FU mesh input, splitting the stream down eight independent paths for the 4x4 mesh that is implemented in the Colt. Note that when a stream splits, the same stream data is sent down each new path, diverging from the original point. In such a case, the addresses in succeeding packets uniquely identify which unit a

particular stream packet is destined for, though it maybe be arriving at four or more units simultaneously.

The crossbar outputs that connect to data ports and the multiplier have been intentionally excluded from this type of broadcast operation. If the broadcast data went to the multiplier, it would pass through and return to the crossbar, making it difficult to control the end path configuration, and impossible to program a path leading from the multiplier output. The reason that this is so is outlined at the end of this section, and pertains to programming paths doubling back on themselves. The outputs from the crossbar that go to data ports also do not respond to broadcast addresses because it makes very little sense to have the same stream come out of more than one port. The ports are the most scarce resource in the system and it would be a waste of system resources to configure more than one port to output the same stream.

| 1st Broadcast Packet | 2nd Broadcast Packet | 3rd Broadcast Packet |
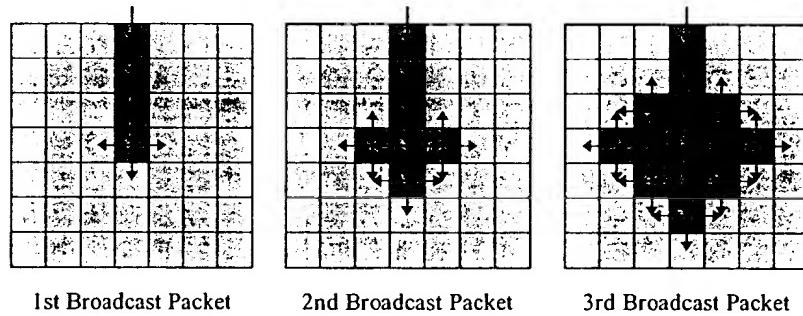
Figure 4.11 - Mesh Broadcast Programming.

In the mesh, a broadcast address can have an explosive effect as well as shown in Figure 4.11. The broadcast address can cause all three of a given FU's neighbors to be programmed with the same information. Three, not four units are programmed because a stream cannot double back on itself. If another broadcast address is in the next stream header packet, 7 FU's

140

will be programmed by it. Like a pebble dropped in a pond, the effect can ripple across the entire

mesh very quickly.
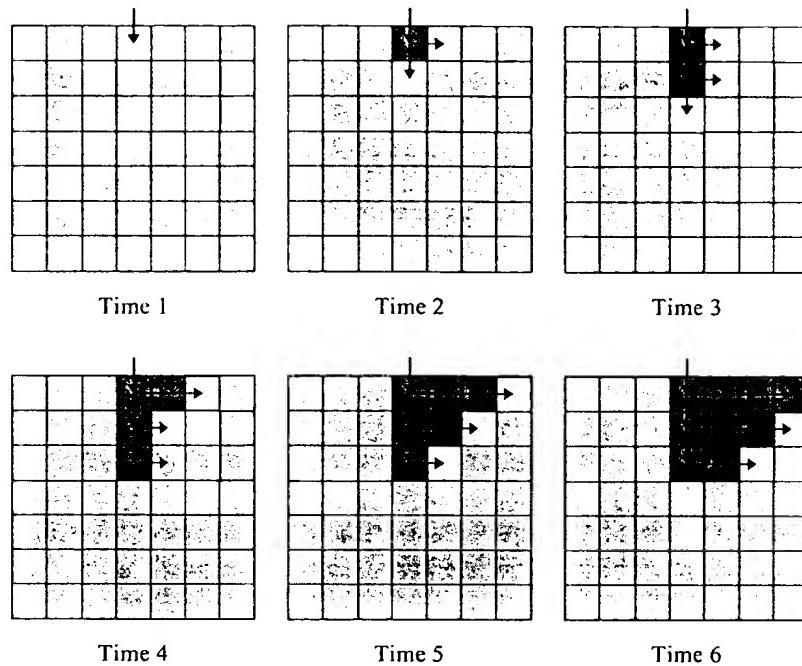
Figure 4.12 - Controlled Mesh Broadcast Programming.

To control this effect, a type of relative address is used for mesh programming in addition

to absolute addressing. The configuration information for an FU contains four bits that indicate

in which direction(s) (north, south, east or west) the rest of the stream header information is to be

forwarded. In this way, the pebble effect can be controlled and directed to sweep in a wave along

a section of the mesh. In Figure 4.12, this controlled broadcast programming is used. Green

indicates FUs that have not been programmed, blue is for FUs that are in the process of being

programmed and red is for FUs that have finished configuration. During time Step 1, the first FU

is programmed to forward stream information to the south and to the east. During time Step 2,

the next header packet programs the next FU to also forward header information to the south and

141

to the east. During time Step 3, the bottom FU is programmed to forward programming information to the east, but the first broadcast packet has already come down the pipeline to the top of the mesh and it begins programming the FU to the east. As the broadcast packets work their way through the pipeline they ripple across the mesh, programming each succeeding FU to forward stream header information to the east. This example adds a level of complexity to that of Figure 4.11 in that the odd timing of packet arrival at various places in the pipeline is taken into account.

It should be noted that if crossbar and data port units are not programmed by the first packet forwarded to them in the stream header, they will not respond to any other packet in that stream. This was done for three reasons. First, it allows the forwarding of a shortened stream header along a pre-existing path. Second, it prevents problems in the crossbar when a stream enters the crossbar, then goes through a unit and comes back through the crossbar again. The third, and most important reason, is that this allows multiple Colt chips to be strung together in a daisy chain, with the same stream running through them all. If units responded more than once to a valid programming address, then the units in the first chip in the chain could respond to packets that were intended for use in another chip further down stream. In the future, it would be preferable to design the entire system in a similar manner to the FUs, such that no unit will forward a stream header until that unit is programmed. After a unit is programmed, however, it would not respond to any additional packets until the end of the stream header has been reached.

4.10 The Valid Bit

The concept of valid and invalid data was introduced in the data port Section 4.6.1. This concept is realized using the valid bit. As discussed earlier, it is difficult to halt the pipeline in the Colt chip due to the timing problem. It would not be impossible, however, using a set of global halt lines that could be tapped by all the units along a given path. The problem is exacerbated when a string of interconnected Colt chips is considered, with a stream flowing off of one and directly onto the next. Then the halt signal would need to not only propagate around the local chip, but over to another chip and around it as well. Some buffering could be employed to allow extra time, but buffers take space and the global halt circuitry would as well. For these reasons, the valid bit is used as a type of feed forward flow control to fill the gaps between valid data. These gaps could be caused by a slow I/O unit, network latency, etc. The valid bit is implemented on chip as a signal that is routed around with each data item. If the valid bit is a logic 1, then the data is valid. If the valid bit is a logic 0, then the data is not valid.

The FUs can be programmed to key off of the valid bit for much of their processing functions. The valid bit that is attached to the output of the FU can be computed in many ways. The first and most obvious method is to logically AND the valid bits of the two input operands to form the valid bit of the result. If either of the two input operands was invalid, the result will be as well. This mode can be used to sum only valid numbers in a stream, or to count valid data easily. To accomplish this, the Left Input Register of an FU can be programmed to only latch valid data. This is then coupled with the loop-back configuration of the FU where the FU's output can be fed back into the Left Input Register without using any routing resources. This

function is useful for summing the products of a dot product operation where the operands (vector elements) are not necessarily guaranteed to arrive at the chip all at once.

The valid bit can be used in other ways as well, such as for conditional execution. In data flow, one way of implementing an IF-THEN statement is to compute both outcomes of the statement, both the IF section and the THEN section. After this, the condition is tested and the correct side of the statement is forwarded on down the stream. The conditional can be used to set the valid bit for the correct side, and reset it for the incorrect side. Both results can then be forwarded to an FU that has been programmed to choose only the valid operand and forward it along. Other methods of using the valid bit will be illustrated in Chapter 5: Example Applications.

5. Example Applications

The previous chapter gave a brief introduction to the architecture and computational abilities of the Colt CCM. This chapter deepens that understanding by detailing the implementation of three different functions on Colt. Starting with an example of the implemention of a dot product, the chapter introduces basic concepts and eccentricities of Colt programming. The implementation of a floating point multiplier is then explored, giving a floating point format that can be easily manipulated using the computing resources of Colt. This example introduces the concept of stream programming and illustrates some of the unique problems that arise due to architectural constraints. The chapter concludes with an example of conditional execution in the form of the calculation of an integer factorial.

5.1 Dot Product Example

A common function in signal processing applications is the dot product. For example, dot products are the core computation in FIR filters. The dot product calculation can be characterized by the evaluation of:

$$f(X) = \sum_{i=0}^{n} a_i \cdot x_i$$

where X is a vector of data values and the individual a_i's come from a vector of weights. There are many methods of implementing this function on the Colt architecture. Which method is the

most optimal choice is dependent on several factors, including the number of available resources, the length of the vectors, the desired speed of execution, the amount of latency that can be tolerated, the choice of floating or fixed point arithmetic and the number of bits of precision used. For purposes of the first example, fixed point arithmetic using the fastest execution method possible will be assumed. One possible implementation is shown in Figure 5.1.
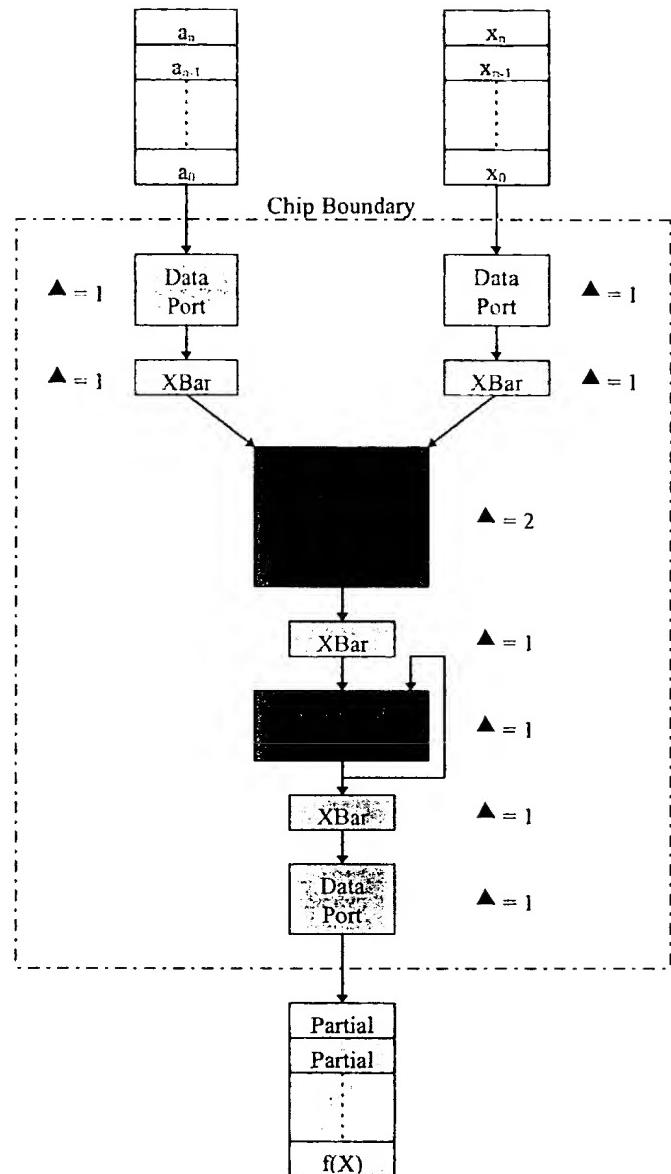
Figure 5.1 - Dot Product Implementation #1.

The stream concept is used to the fullest in this implementation. The vector elements are

stored sequentially in the memory system and are presented to the data ports in order. The data

values pass from the data ports through the crossbar to a built-in multiplier. The pipelined

147

multiplier requires two cycles to evaluate each product and then passes it on through the crossbar

to an FU that has been configured as a summing node. The partial sums are sent through the

crossbar to a data port, and from there the partial sums are written into memory in sequential

order. The last value written out will be the final sum. Note that the delta delay for each

pipelined stage is given in clock cycles next to each component. There are several operating

details that are hidden by this diagram; the most significant of these are addressed below.

5.1.1 Data Port Synchronization
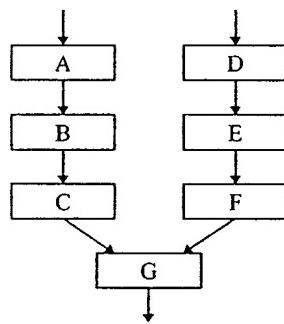
Figure 5.2 - Pipeline Halting Problem.

Obviously, in order for the pipeline calculation to be correct, the a_i's must be matched

with the proper x_i's. All data flow through the chip is pipelined, with no real concept of registers

or temporary storage. Hence, any time that the flow of data through the chip is stopped the entire

pipeline must be shut down and a great deal of efficiency is lost. The objective then is to keep

the pipeline full and running at all times. Further, the practical implications of trying to stop a

running pipeline are rather daunting. Consider the communications difficulties for a unit down

stream in the pipeline that causes a pipeline halt. Such a situation is illustrated in Figure 5.2. In

this picture, the shaded boxes represent FUs that contain valid data ready to be passed to the next

148

unit in the pipeline. Note that FU C does not yet contain valid data. During the next time step, FU G can accept valid data from FU F, but not from FU C. Ideally, a flag from FU G would be propagated back to FU F to indicate that the pipeline should halt until valid data is ready at FU C. This would work well if the signal only needed to be sent to FU F, but the halt signal must also be propagated back to FUs E and D as well. These FUs may or may not be physically close to FU G on the chip, yet the halt signal must be propagated to all FUs in the right half of the graph within one clock cycle. Guaranteeing this condition in hardware is a problem that is only exacerbated as the length of the pipeline increases.

For these reasons, no mechanism of stopping the pipeline has been incorporated into the design. Once the pipeline is started it must run, each unit accepting a new data value every clock cycle, through completion of the last calculation of the last data value in the stream. This is not to say that no flow control exists on Colt, and, indeed, limited flow control can be provided using the valid bit and the different modes offered by the data ports.

The Stream Controller must be programmed to deliver the proper operands to the data ports that are used for a particular data flow graph page. A memory ready signal is sent from the Stream Controller to each data port as each memory pipe is connected. When all data ports are connected to memory pipes and all are ready to read/write data, a signal is generated within the Colt chip and sent to all data ports simultaneously. Depending on the programmed mode of the data ports, different flow control measures are taken. Raw Mode makes no attempt at synchronizing the inputs to the various ports, while Synchronization and Loop Modes do provide this functionality. However, in all cases, once the movement of data through the chip has begun, it cannot stop.

149

Synchronization of the injection of new data into the chip is controlled by a *SynchronizationReadyOut* signal that is generated by each data port. All data ports for a given page are programmed with a mask indicating with which of the five other data ports they should be synchronized. When data is available at all input ports and the Stream Controllers are ready to accept data at the output ports, all ports involved will assert *SynchronizationReadyOut*. These signals will be masked by each data port to generate a *SynchronizationReadyIn* signal, indicating that new data may be allowed to enter the chip.

For an input port in Synchronization or Loop Modes, assertion of *SynchronizationReadyIn* will cause the port to begin accepting data at the rate of one 16-bit word per clock pulse. Up until this point the data port has been sending data onto the chip with the valid bit set to 0. The data will now be sent with the valid bit set to 1 until the Transmit signal is de-asserted by the memory pipe. Whenever the Transmit signal is de-asserted, the data port will enter the waiting state and the process begins again. The number of data items in the input stream is controlled solely by the Stream Controller through the use of the Transmit signal. Output data ports function slightly differently because it is difficult to determine when all the data has gone through the entire on-chip pipeline. The output data port will do a memory write for every data item it receives for which the valid bit is a 1. All invalid data is ignored and effectively thrown away.

A data counter could have been designed into the data port so that the number of valid operands could be counted and the output data port could reset itself; however, such a counter puts a fundamental limitation on the data stream size. One possible mode of operation for the Colt chip is to string a series of them together to form a larger pipeline and run them with a static configuration. A counter would no doubt overflow in a short period of time in such a situation.

150

Further, the input data ports already require the Stream Controller to contain such a counter, which could no doubt be re-used in this circumstance.

5.1.2 Summing Node Initialization
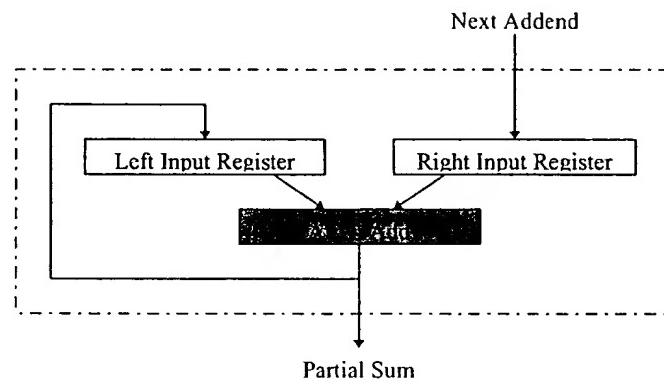
Figure 5.3 - Summing FU Logical Diagram.

Another hidden detail in Figure 5.1 is the operation of the *Summing FU*. Figure 5.3 shows a logical depiction of the physical configuration of the *Summing FU*. The *Next Addend* is latched into the *Right Input Register* on each clock pulse. The *Right Input Register* is added to the *Left Input Register* and the new partial sum is latched back into the *Left Input Register*; hence the *Left Input Register* acts as an accumulator. The partial sum is also sent to the primary output of the FU. The difficulty arises first in initializing the *Left Input Register* to zero and then in making sure that accumulation doesn't start until data has propagated to this point in the pipeline. Only the *Left Input Register* can be initialized to a desired value through the FU programming information.

The problem of determining when accumulation of partial sums should begin is more difficult. In the Colt, it is solved by the use of the valid bit. The valid bit acts as a flag to

151

indicate when a given data item is valid (a good data item) or when it is invalid (un-initialized garbage). This bit is attached to the left end of all data operands and is 1 if the data is valid or 0 if the data is invalid. In the Colt chip, all valid data is processed and invalid data may or may not be processed. The *Left Input Register* can be programmed to either accept or reject invalid data.

When the page shown in Figure 5.1 is initialized, the valid bits for all operands, except for the *Summing FU's Left Input Register*, will be initialized to 0. Hence, all data initially in the pipe is flagged as being invalid. Between the time when programming is completed and when data ports begin to receive good data from off chip, the pipeline remains in this initialized state because the data coming from the input data ports has the valid bit set to 0. This invalid data is propagated through the networks intact. When the data arrives at the multiplier the valid bits for the two multiplicands are ANDed together to produce the valid bit for the product. Hence, the output of the multiplier will only be valid if both input operands are valid. Since both input operands are invalid the multiplier output will be invalid as well. The invalid product reaches the *Right Input Register* of the *Summing FU* and since the *Right Input Register* always latches, the invalid data is latched. Within the *Summing FU*, the invalid data will be added to the *Left Input Register* (which currently contains a valid 16-bit operand of value zero) and an output will be produced. However, the valid bit for the sum will be 0 because, like the multiplier, the *Summing FU* should be set to AND the valid bits for the *Left* and *Right Input Registers* to produce the valid bit for the sum. In this case, the valid bit for the *Right Input Register* is 0 and the valid bit for the *Left Input Register* is 1. Because the *Left Input Register* is programmed to only latch valid data, the new sum will not be latched and the initial partial sum of zero will be preserved in the *Left Input Register*.

152

When the data ports receive good data the valid bit for each operand is set to 1. The good data propagates through the pipeline to the multiplier. Both data ports are started simultaneously and both pipeline path lengths to the multiplier data inputs are equal so that good data arrives at both multiplier inputs simultaneously. The valid bit for the multiplier output is calculated by ANDing the two valid bits together and this time the result is 1, so the multiplier output will be valid. The valid product is routed to the *Summing FU* where it is latched by the *Right Input Register*. At this point, there is valid data in both the *Left* and *Right Input Register*s so when the valid bit for the sum is calculated it will be a 1. Hence, the sum will be valid and it will be latched into the *Left Input Register* as the first partial sum. As more valid data comes down the pipeline to the *Summing FU*, it will be added to the partial sum and latched into the *Left Input Register* in the same way. Note that on every clock pulse on which valid data arrives at the *Right Input Register*, valid data will be sent on to the output data port.

5.1.3 Output Data Port Control

It may or may not be desirable for all partial sums to be written out to the Stream Controller. At the least, the number of partial sums will be known and the last one can be taken from the Stream Controller and the rest ignored. More likely, it would be better to only write out the final sum and not write out all the intermediate results. It is possible to modify the example so that only the final sum is written out to the Stream Controller. This modification is shown in Figure 5.4.

153
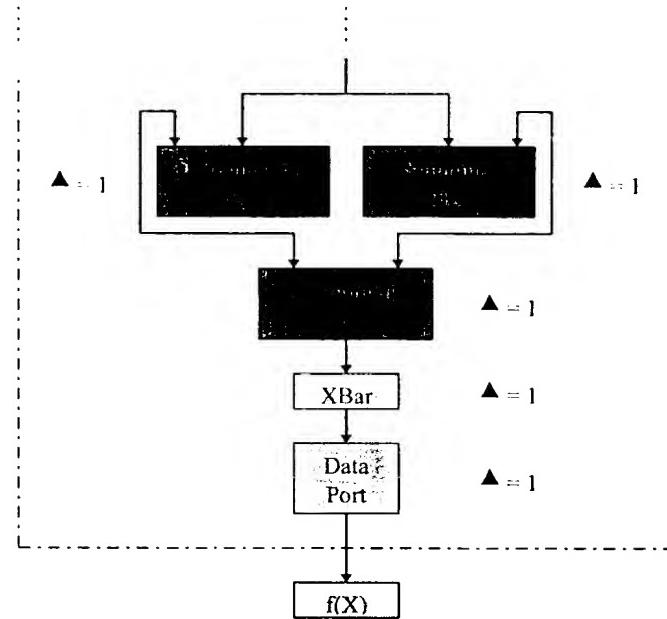
Figure 5.4 - Data Counting Selection Scheme.

Here, two more FUs have been added to the graph in order to effectively select only the

last data value in the set so that only it is written out to the memory pipe. The *Decrementing FU*

logical configuration is shown in Figure 5.5.
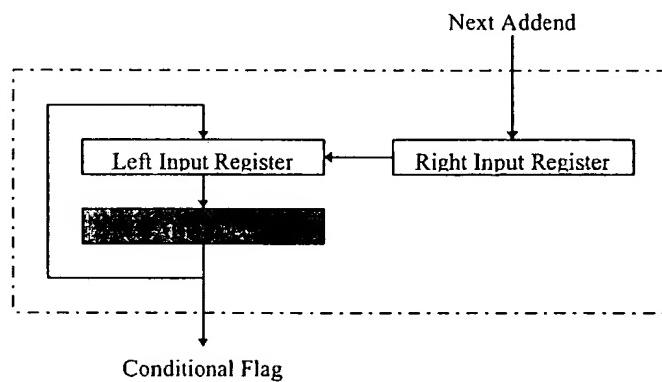
Figure 5.5 - Decrementing FU.

As seen in Figure 5.5, the FU is again configured to loop back the result of the ALU operation. The *Right Input Register* always latches in the *Next Addend* (the same as that fed into the *Summing FU*) on every clock pulse. The *Left Input Register* is configured to only latch valid data and it is initialized with the number of partial sums that will be generated before the final result is completed. The valid bit is initially set in the *Left Input Register*. As before, the valid bit for the FU result is the logical AND of the *Left Input Register* and the *Right Input Register*; hence, the decremented value of the *Left Input Register* will only be latched when the value latched into the *Right Input Register* is valid. This dependency is shown by the arrow from the *Right Input Register* to the *Left Input Register*. This effectively counts the number of valid products that have arrived at the *Summing FU*. When the last product enters the *Summing FU*, the count will have gone negative and so the sign bit coming out of the ALU will indicate a negative value. The sign bit can be connected to the *Conditional Flag* leaving the FU, and this is connected to the *Combining FU* through the network. The *Conditional Flag* will be 0 until the last product arrives, after which it will be a 1.

The *Combining FU* receives the *Conditional Flag* from the *Decrementing FU* and the partial sum from the *Summing FU*. The *Conditional Flag* is ANDed with the valid bit from the partial sum to produce a new valid bit. The partial sum itself goes through unchanged. Since the *Conditional Flag* is 0 until the last partial sum is produced, after which it is 1, the output of the *Combining FU* will not be valid until the last partial sum is produced. Further, after the last partial sum is produced, no more valid data will enter the *Summing FU*; hence the last valid data to come out of the *Summing FU* will be the last partial sum. Therefore, the last partial sum will

155

be the one and only valid data item that will leave the *Combining FU*, and so the last partial sum will be the one and only data item to be written to the memory pipe by the output data port.

5.1.4 Page Re-initialization

After a data set has been processed by a page there may be a need to run a second data set through the same page without complete reconfiguration. This is possible in the Colt chip. For pages that contain no concept of state, the only concern is data port synchronization. Synchronization of the beginning of data should be solved by the Stream Controllers that need to be designed so that they will allow the stream headers to enter the chip independently, but will then restrain the data sections of the streams until all are ready. The Stream Controllers will then assert a start signal, allowing the data sections of all streams to enter the chip simultaneously. Further data synchronization can be handled by the Colt.

For pages containing some sort of state information, such as the example in Figure 5.4, some re-programming of the FUs will be necessary. In the example, the *Decrementing FU* and the *Summing FU* would both need to be re-programmed so that the constant values and the valid bits could be reset to the correct values. The number of nodes that would require reconfiguration after a "run" should be small. Foreseeing this possibility, the FUs have been designed to support single word programming so that FUs that require no re-initialization can be quickly skipped over by a shortened packet in the revised stream header. This shortened packet contains only one word of programming information for these FUs instead of the normal eight words. When the revised stream header arrives at an FU that requires re-initialization, a full eight word packet can

be used to reinitialize it. Thus, the number of clock cycles required to retrace the original path of the stream through the chip for purposes of re-initialization is significantly reduced.

It should be noted at this point that with proper design the *Summing FU* would not require re-initialization. A mode has been included for the *Left Input Register* to handle this situation in which the register can be conditionally zeroed out by normal stream execution. Further, with the addition of one more FU to hold and compare the vector length constant, the *Decrementing FU* structure could also be configured so that no external re-initialization would be required.

5.2 Floating Point Multiplication Example

The Colt chip will be also be required to perform floating point operations. As is so often the case, these operations are discouraged because of the additional time and resources required to perform them. However, certain provisions have been made within the Functional Units to ease the burden when floating point mathematics are required by the problem at hand. In order to demonstrate these capabilities, the example of floating point multiplication will be addressed. There are three main steps to consider: mantissa multiplication, addition of exponents and re-normalization.

5.2.1 Format

1 bit	15 bits	16 bits
Sign	2's Complement Exponent	Mantissa
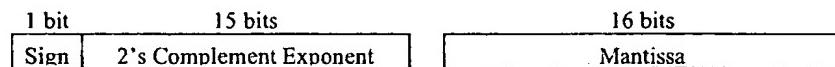
Figure 5.6 - Floating Point Format.

157

As can be seen in Figure 5.6, the floating point number is split into two separate 16-bit words so that they can be easily split into two separate data streams for processing. Floating point representations often use a biased exponent representation instead of a true two's complement form so that an unsigned comparator can be used to determine relative numeric magnitudes; however, signed and unsigned comparators have the same cost on the Colt chip, so the advantage of biased arithmetic is negated. Further, biasing forces the bias to be added and subtracted during processing, which would require added resources to implement on the Colt chip. Another often used floating point technique is the assumption of an implied binary digit 1 to the left of bit 15 of the mantissa. This technique will not be used here for the sake of simplicity. It will be assumed that the mantissa is always normalized so that the leading bit is always a 1. The sign bit for the represented quantity is stored in bit 15 of the exponent, and for the sake of convention, it will be assumed that 1 indicates a negative number and 0 indicates a positive.

The overall effect is to create a non-standard floating point format that would require post processing before the results could be used in an IEEE standard computer. The bulk of processing should be done using the Colt chip so that at most only a conversion at the beginning and end of computation would be performed. Further, in many applications floating point arithmetic is only needed to calculate intermediate results. In these cases, no external conversions would be required since all computations would be performed internally. Finally, this is only one possible format and the Colt programmer can conform to whatever format is desired.

5.2.2 Implementation

The full floating point multiplier is shown in Figure 5.7. Two floating point values are input to the Colt. The "Left" value is injected through ports 1 and 2 and the "Right" value is injected through ports 3 and 4. The final result comes out of ports 5 and 6. Solid lines indicate that a local connection is used. A dotted line indicates that the Skip Bus is used to pass the value.
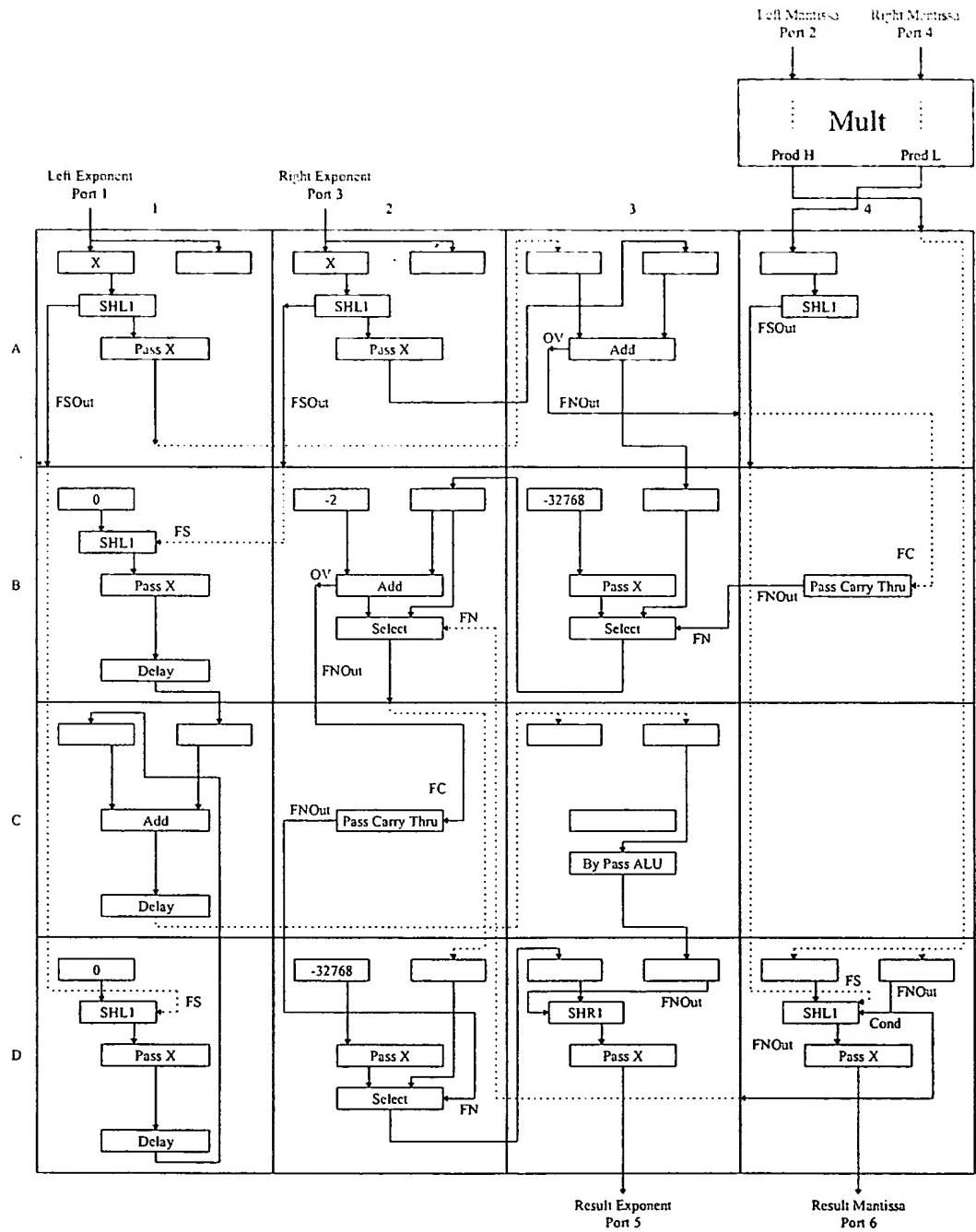
Figure 5.7 - Floating Point Multiplier Implementation.

160

The path taken by the two mantissa values is the simplest and so will be considered first. The mantissas enter through *Ports 2* and *4* and proceed directly to the multiplier. After a two clock cycle latency a 32-bit result is produced. Only 16-bits of this result will be propagated through to the final result and the remaining bits will be discarded as lost precision. However, after multiplication, the 32-bit value must be re-normalized so that the leading bit of the resultant mantissa is a 1. This problem is simplified somewhat because the two input values are already normalized; thus, when multiplied together, the most significant 1 in the binary result will be either in the leading position (bit 31) of the result or one position to the right (bit 30). Thus, the leading bit of the 32-bit mantissa can be tested and if the result is 0, the mantissa needs to be shifted once to the left and the exponent must be decremented by 1.

The right hand column of the mesh is configured to perform the re-normalization function (FUs A4, B4, C4 and D4). The low word of the 32-bit result is latched into the *Left Input Register* of FU A4, where it is shifted once to the left. The bit shifted out may need to be shifted into the low bit position of the high word of the 32-bit result if normalization is required. This shifted bit is sent down the *Skip Bus* to FU D4, where it is supplied to the *Barrel Shifter* as the bit to be shifted in if re-normalization is required. The high word of the 32-bit result is latched into both input registers of FU D4 at the same time that the low word is latched into FU A4. Notice that the high word traverses the entire height of the mesh using the *Skip Bus* to arrive at FU D4 at the beginning of the same clock pulse. The value is latched into both input registers so that it can be run through the *Barrel Shifter* from the *Left Input Register* and so that the high bit of the value can be tested using the *FNOut* flag, which is connected to bit 15 of the *Right Input Register*. *FNOut* is connected to the conditional logic of the *Barrel Shifter* so that if the high bit of the

Right Input Register is a 0, the shift will be performed. Otherwise, no shift is required. The ALU P, G and R bit vectors are set to pass the value in the *Left Input Register* (X) straight through without change. This value is the final mantissa of the result.

The exponents of the floating point values take a more convoluted path through the mesh. Each sign bit must first be separated from its exponent so that the sign bits may be XORed together to produce the sign bit of the result. The exponents must be added together and then tested to determine if an arithmetic overflow occurred. If an arithmetic overflow is detected, the number is either to large or to small to be represented and the exponent of the result will be reset to the smallest possible value. After testing for overflow, the exponent may be decremented once more if the mantissa multiplication result requires re-normalization. This, too, is tested for arithmetic underflow, and is adjusted back to the smallest possible exponent if it occurs. Then the sign and exponent of the final result are reunited and the upper word of the final result is sent out *Port 5*.

The path that the signs take through the mesh begins at FUs A1 and A2. The left exponent word is latched into both input registers of FU A1 and the right exponent word is likewise latched into FU A2. In this case, the values are latched into both registers as a convenient way of producing an accurate valid bit for the result. By latching into both registers, the valid bit of the result can be programmed to be the logical AND of the two valid bits in both the input registers. The sign bit is extracted from each word by the *Barrel Shifter*, being set to unconditionally shift once to the left. The bit shifted out is the sign bit of each value. The sign bit for the left value is sent along *FS Skip Bus* segments to FU D1, while the sign bit for the right value is sent on *FS Skip Bus* segments to FU B1. At these FUs, the sign bit is unconditionally

shifted left by 1 through the *FS* flag into the least significant position of the value that is input to the ALU. This value is latched into the optional output delay of the FU. Note that the *Left Input Register* in both FU B1 and D1 has been programmed with a constant 0 value so that the result that is latched into the delay will be either decimal 1 or 0; although all that needs to be preserved at this point is bit 0. The sign bits must be XORed together to produce the sign bit of the result. This is accomplished in FU C1, where the ALU has been configured as an adder. Because the sign bits are in the least significant position (bit 0) and the carry in is programmed to be a constant 0, this addition has the effect of XORing the sign bits. To equalize the path delays between this computation and the exponent computations, the resultant sign bit is then latched into the optional output delay of the same FU. One more delay is required to equalize the path lengths, so the resultant sign is latched into FU C3 by traversing the *Skip Bus* to that point. It then by-passes the ALU of FU C3 using the *Conditional Unit* and is latched into FU D3, where it is united with the exponent of the final result.

The path of the exponents also begins in FUs A1 and A2. After the unconditional shift left by the *Barrel Shifter*, the exponents pass through the ALU, which has been programmed to pass the value from the *Barrel Shifter* (labeled X) unchanged. The two values then arrive at FU A3 simultaneously, and are latched into the input registers. The exponent values are then added in FU A3, and the *FNOut* flag is set to detect an arithmetic overflow from the ALU. The result of the exponent addition is latched into the *Right Input Register* of FU B3. If an arithmetic overflow occurred, the value must be reset to the smallest possible exponent (-32768). To accomplish this, -32768 is programmed as a constant into the *Left Input Register*, and the *Barrel Shifter* and ALU are configured to pass the *Left Input Register* through unchanged. Then the

Conditional Unit is used to select between the constant in the *Left Input Register* and the computed sum of the two exponents.

The determination of which will be selected is controlled by the *FN* flag. If an overflow was detected during the addition, this would have been signaled on the *FNOut* flag of FU A3 during the previous clock pulse. In order to store the value of that flag at that time, it was routed to FU B4, where it was latched into the carry flag of the ALU. The ALU of FU B4 is configured to pass the carry straight through, where it is carried by the *FNOut* flag to FU B3 so that it can be used to control the selection of either the result of the exponent addition or the constant. The diversion through FU B4 is necessary because a one clock cycle delay is needed for the signal, just as the result of the addition will experience when it is latched into FU B3. Routing through FU B4 adds the same delay to the overflow flag so that the timing matches.

After the overflow flag selects between either the constant or the computed exponential sum, the result is routed to FU B2. At FU B2, the exponent value is conditionally decremented to allow for a possible re-normalization of the mantissa value. Note that here the selection is between the addition of -2 to the exponential sum or passing the sum through unchanged. The constant -2 is programmed into the *Left Input Register* because the exponent is currently shifted left once, so that adding negative two at this point is equivalent to decrementing the resultant exponent by 1. The ALU is programmed for a standard addition of the exponential sum and the constant -2 and the *Conditional Unit* is controlled by the same flag that controls the conditional shifting of the mantissa if re-normalization is required.

Again, underflow detection has been implemented to reset the value of the exponent if the decrement causes an underflow to occur. The actual reset occurs in FU D2 and the overflow flag

164

is routed through FU C2 to provide the necessary delay. Note that there is a small bug in this implementation. If FU D2 detects that the decrement would cause an arithmetic underflow of the exponent, the result is reset to the smallest possible exponent regardless of whether or not the decrement was actually performed. This was considered to be a minor problem because the exponents involved are at the extreme lower range of the possible values, though the problem could be solved with a slightly different implementation.

The final exponent value is routed to FU D3, where it is united with the final sign value. This is done by unconditionally shifting the exponent right by 1 and setting the shift in value to be the least significant bit of the *Right Input Register*, using the *FNOut* flag. The ALU is then programmed to pass the value from the Barrel Shifter through unchanged, and the final sign-exponent pair is sent to *Port 5*.

Note that there are several critical timing points within this configuration. The first is the timing of the arrival of the sign bits at FU C1. Both sign bits must experience the same net delay in arriving at that point. If not, the misaligned sign bits will be XORed together and an incorrect result will be computed. Note that upon arriving at the input registers of FUs A1 and A2, both sign bits then experience two clock delays before being added together: one at the optional output delay of FUs B1 and D1 and the second at the *Left* and *Right Input Registers* of FU C1. The next critical timing occurs at FU B2 where the flag that is used to control the mantissa alignment process must arrive at this unit at the same time as the exponent corresponding to that mantissa. The mantissa path (and delays) through the chip has been: data port (1), crossbar(1), multiplier (2), crossbar(1), FUs A4 and FU D4 (1), for a total of 6 clock pulses. The exponent path has been: data port (1), crossbar(1), FUs A1 and A2 (1), FU A3 (1), FU B3 (1) and FU B2 (1), for a

165

total of 6 clock pulses. Thus, the correct mantissa result will be influencing the correct exponent result. The last rendezvous that must occur is the reunion of the computed sign bit and the computed exponent. The path of the sign bit has been: data port (1), crossbar (1), FU A1 and A2 (1), FU B1 and D1 (1), FU C1 (2) and FU C3 (1), for a total of 7 clock pulses. The path of the exponent has been: data port (1), crossbar (1), FU A1 and A2 (1), FU A3 (1), FU B3 (1), FU B2 (1) and FU D2 (1), for a total of 7 clock pulses. Thus, both the exponent and the sign bit experience 7 clock pulses of latency before converging on FU D3 for final processing.

5.2.3 Programming Method
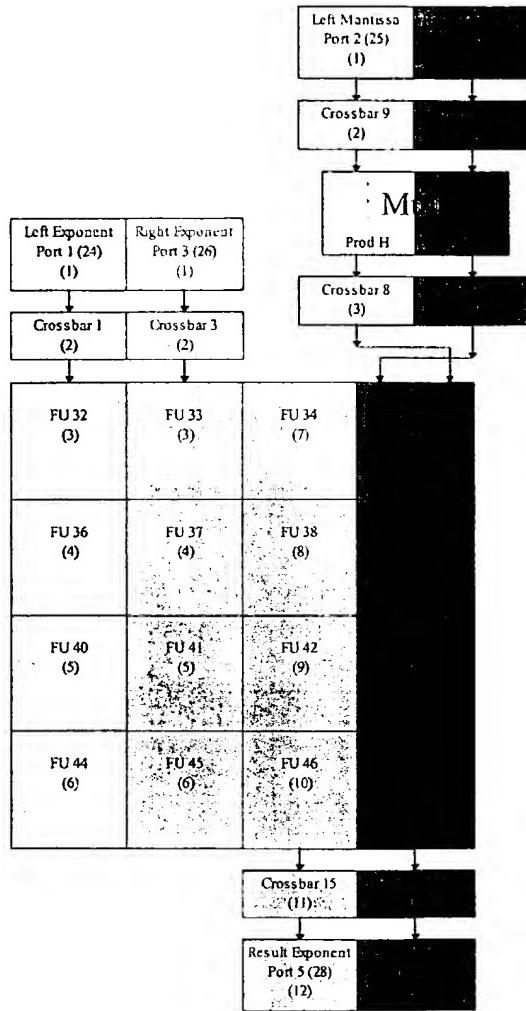
Figure 5.8 - Floating Point Multiplier Programming Method.

The programming path followed by each of the four streams is shown in Figure 5.8. The four streams are color coded for clarification. Each box indicates a unit that must be passed through by the stream. The number in parenthesis on the last line of each box indicates the order in which the unit is programmed by the stream. For example, the blue column down the left

167

hand side shows *Data Port 1* being programmed, followed by the crossbar and then the four leftmost FUs in the mesh. The crossbar and FU labels are followed by a number indicating the address that must be used to program that unit. The ports are referred to by number, but their unit addresses are shown in parenthesis after the port number.

There are several interesting points shown by Figure 5.8. The first is that the multiplier is split into two halves: one stream enters the left input and exits out the high word of the result, the other enters the right input and exits out the low word of the result. There is nothing in the multiplier to configure so that all programming information is simply passed straight through in this manner. The yellow stream starts as the left mantissa value and first programs *Port 2*, followed by the crossbar, exiting through address 9 which will enter the left input of the multiplier and then exit through high word of the result. The high word of the result is to be routed to the *Skip Bus* input of the right most mesh column; thus, crossbar output 8 is then configured as the crossbar exit point. This is the end of the yellow stream since configuration information cannot be sent through the *Skip Bus*.

The red stream containing the right mantissa value configures the rest of the computational path used for the mantissa. It begins at *Port 4* and needs to enter the right multiplier input so it programs the crossbar for *Output 10*. Passing through the multiplier and out the low word of the result, the stream programs crossbar *Output 7* so that it will enter the local north input of the rightmost mesh column. It then proceeds to program the four eastern FUs in order and emerges out the local output at the bottom of the column. From there the final result is to be output from *Port 6*, so it programs the crossbar to route itself through *Output 16* so that it arrives at *Port 6*. The red stream then programs *Port 6* and exits the chip.

168

The green stream is the last interesting point of this configuration. Notice that it must configure two columns of the mesh, one of which does not exit out the bottom. To accomplish this, the green stream (the right exponent) enters through *Port 3* and programs the crossbar for output address 3 so that it arrives at the local north input of the second column. It then programs FU 33 to forward programming information to the south and to the east. The following three stream packets are addressed uniquely for FUs 37, 41 and 45 in succession. Meanwhile, these same packets are forwarded to FU 34, but because the addresses do not match it does not respond. The packet to configure FU 34 is placed after the packet for FU 45 in the stream. FU 34 responds to that packet because it is uniquely addressed for it and so the third column of the mesh is programmed by packets uniquely addressed for FUs 38, 42 and 46, respectively. After FU 46 is programmed, the stream exits the bottom of the mesh and programs crossbar output 15 so that it is routed to *Data Port 5*, which it then configures before exiting the chip.

5.2.4 Improved Implementation

The path equalization process is a key component of design when using the Colt, and is arguably the trickiest as well. Note that in the configuration shown in Figure 5.7 the exponent word of the final result will emerge from the chip several clock pulses behind the corresponding mantissa word due to differing path lengths. The mantissa word follows the path: data port (1), crossbar (1), multiplier (2), crossbar (1), FU D4 (1), crossbar (1), data port (1), for a total of 8 clock cycles of latency. The exponent follows the path: data port (1), crossbar (1), FU A1 (1), FU A3 (1), FU B3 (1), FU B2 (1), FU D2 (1), FU D3 (1), crossbar (1), data port (1), for a total of 10 clock cycles of delay. It is possible to correct this difference by inserting two additional clock

169

delays in the path of the mantissa. In many situations, this should not be necessary since the valid bit has been used throughout the design to flag good and bad data. The separate valid bits are kept with each word wide result from each FU, and these are propagated all the way to the output data ports. The Colt chip will only write valid data from an output port and hence the final misalignment can be handled by the Stream Controllers before being passed on to the rest of the system, if indeed it is a problem at all.

One situation where this design could become problematic; however, would be the case where multiple Colt chips are directly connected together. Succeeding chips could be configured to allow for the difference in latencies through this design, but after several chips, each with different execution path lengths, the difference may become to great to cope with. Further, the design shown in Figure 5.7 makes extensive use of long Skip Bus paths. While there is nothing theoretically wrong with this approach, in practice it may require that the clock rate used for the chip be slowed somewhat to allow for the extra propagation delay. Shortening these paths would be beneficial to ensuring the highest possible clock rate.
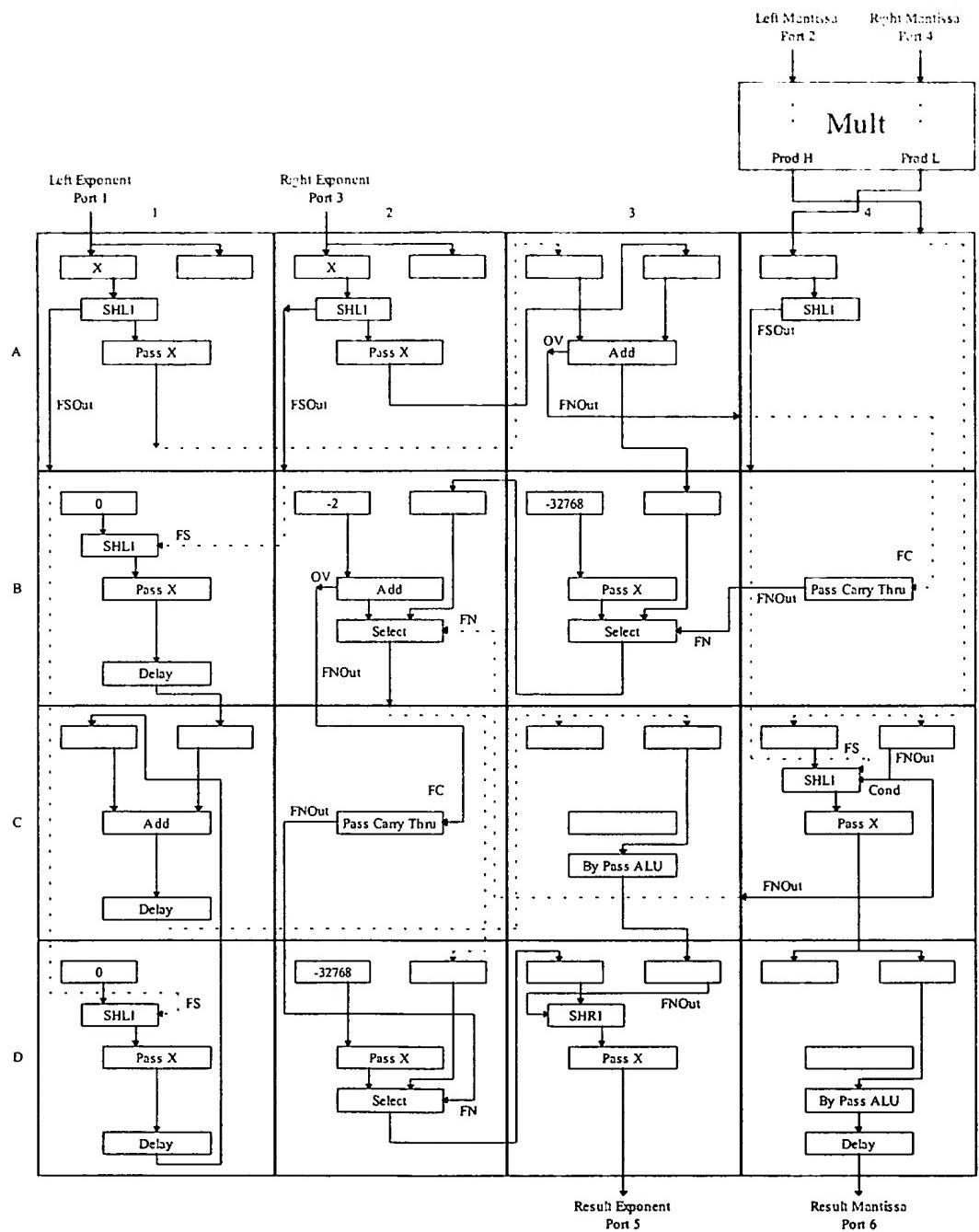
Figure 5.9 - Improved Floating Point Multiplier Design.

171

Figure 5.9 shows the improved design. The first goal was to insert two additional clock delays in the path that the mantissa follows through the chip. Care must be taken as to where the delays are inserted in order to avoid disturbing the timing of the equalized paths discussed above. The only critical path length involved in the computation of the mantissa is up to the point of re-normalization. Thus, the two additional delays can be inserted after this point without disturbing the timing. This was done by moving the normalization operation previously performed in FU D4 up to FU C4. This also has the fortunate effect of shortening the length of the *Skip Bus* paths needed to get the input operands to the computation. It also shortens the path length of the *FNOut* flag as it goes to FU B2. The two additional delays have been inserted by the new FU D4 by simply latching the final mantissa into both input registers. The *Left Input Register* is routed around the ALU using the *Conditional Unit* followed by the *Optional Output Delay*, giving two clock cycles of latency before being sent on to the output port.

5.3 Conditional Execution

The implementation of conditional execution structures within the Colt architecture can be confusing because of the deep level of understanding it requires. A full understanding of the valid bit, conditional logic and the data port modes is required. In an attempt to clarify the design process, the calculation of a 16-bit integer factorial function will be detailed in this section as an example.

172

5.3.1 Factorial Concept
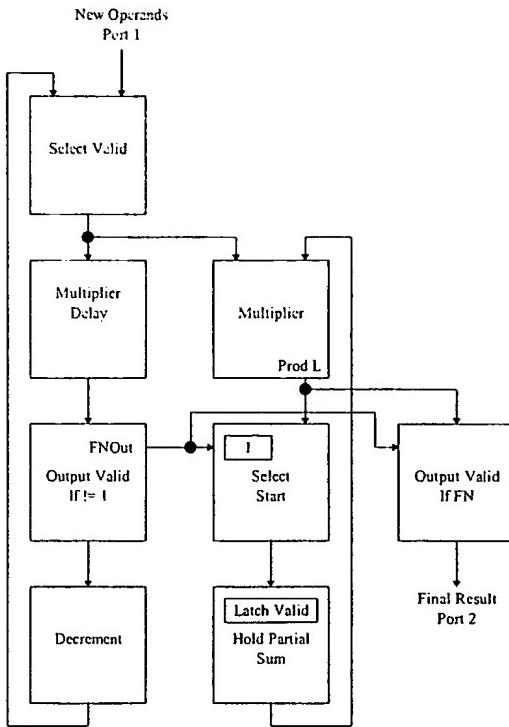
Figure 5.10 - Conceptual Factorial Implementation.

The conceptual design of the factorial function is shown in Figure 5.10. The calculation

of the factorial function is a data dependent looping problem because the number of

multiplicative iterations required is dependent on the input operand. The implementation shown

can really be considered as two separate executing processes that must exchange data at timed

intervals.

The first process is shown by the functional blocks in the left column of the figure. In

these blocks, the input operand is decremented until it reaches 1. Because this function has an

unknown execution time, a collision free pipeline cannot be constructed. The alternative is to

use the data ports in Loop Mode so that they will guarantee that only one valid operand exists in the pipeline at any given time. The *Select Valid* block shown acts as a multiplexer that will switch to forward whichever of the two inputs are flagged as being valid. The pipeline is initialized with only invalid data and then the first operand is injected. While that operand is being processed, it circulates around the loop and the *Select Valid* block will continue selecting it until the computation is complete. The original input value circulates in the loop shown on the left and each time it does so it is decremented by one. When the value is decremented to 1, the *Output Valid If != 1* block will cause the operand to become invalid. When a valid result is written from *Data Port 2*, it will trigger the injection of a new valid operand from *Data Port 1*, and because the previous value is now invalid, the *Select Valid* block will select the new operand and the process begins again.

The multiplication calculations are performed by the process that executes in the loop shown on the right side of Figure 5.10. The *Latch Valid* register shown is used to hold the partial product at each iteration of the calculation and is initialized with a valid 1 at configuration time. The *Multiplier* will not output a valid result unless both inputs to it are valid. That will only happen when the operand value being decremented circulates through to the output of the *Select Valid* block. At that time, the *Multiplier* will produce a valid result, which will be sent to the *Select Start* and *Output Valid If FN* blocks. The *Select Start* block will always forward the results of the multiplier unless the original operand has been decremented to 1, and the calculation is complete. If the calculation is not finished, the new partial product is forwarded to the *Hold Partial Sum* block and then to the *Multiplier* where the process will begin again when the original operand re-circulates. If the calculation has finished, then the *Select Start* block will

forward a valid 1 to the *Hold Partial Sum* block so that it will be initialized for the next calculation. At the same time, the *Output Valid If FN* block will receive the fully computed value and will send it out *Data Port 2*.

5.3.2 Mapping Strategies

With the general functionality established, implementing this function on the Colt chip becomes a matter of equalizing execution path delays so that corresponding values meet at the right units at the right times. There are two critical timing points. The first is that the output of the *Multiplier* arrives at the *Select Start* and *Output Valid If FN* blocks at the same time that the operand value is latched into the *Output Valid If !=* 1 block. Second, the new value from *Hold Partial Sum* must arrive at the *Multiplier* at the same time or earlier than the decremented version of the original operand.

In the Colt, the *Multiplier* has a direct connection to the crossbar. This gives guidelines for the placement of the functional blocks to FUs in the mesh since it would be logical to place an FU that supplies an input to the *Multiplier* at the bottom of the mesh and to place an FU receiving a *Multiplier* output at the top of the mesh. Also, the delay through the *Multiplier* is a total of four clock cycles: one to traverse the crossbar to the multiplier, two for the multiplier itself and one to traverse the crossbar back to the mesh. This delay must be duplicated by the *Multiplier Delay* block. One possible implementation is shown below in Figure 5.11.
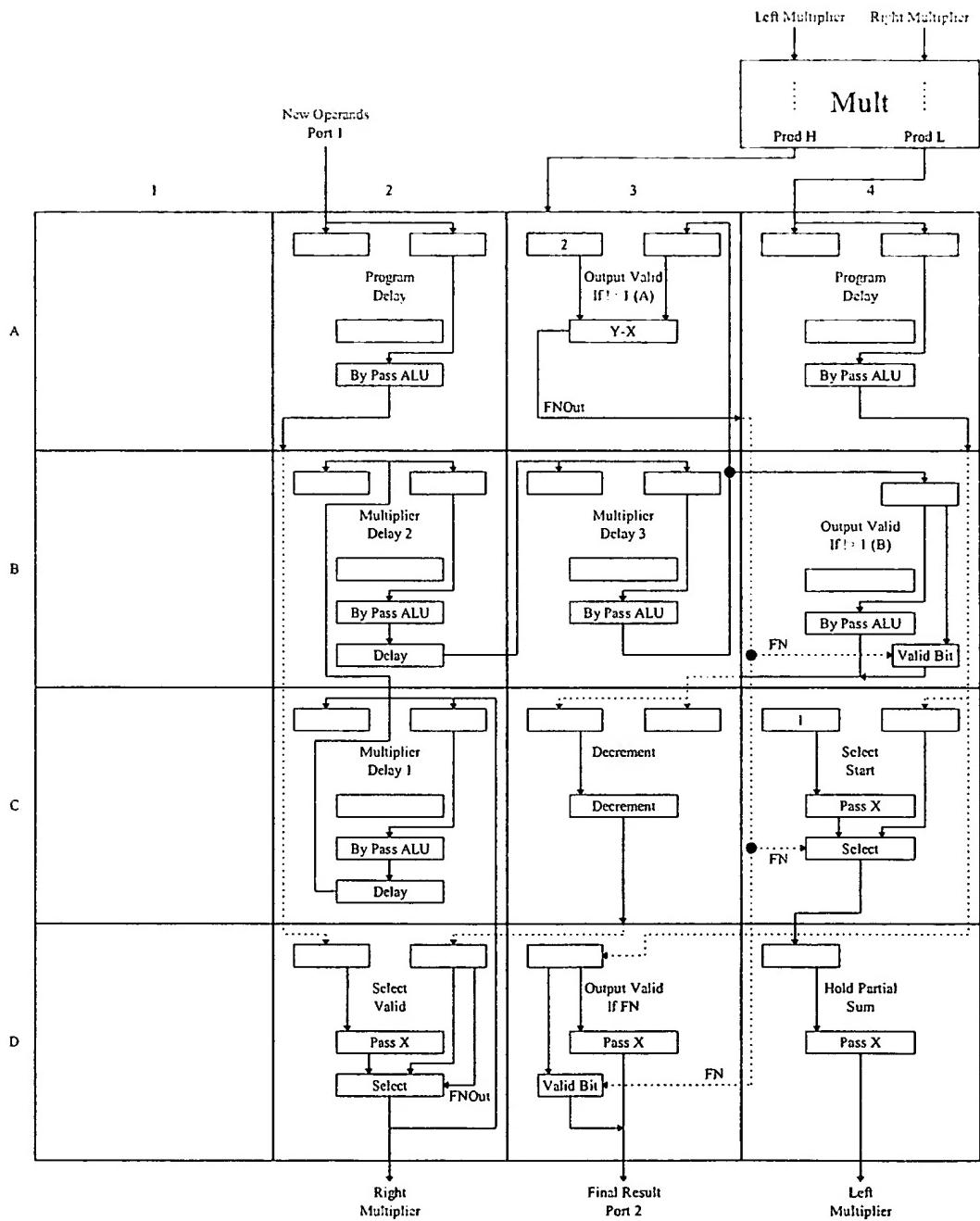
Figure 5.11 - Colt Factorial Implementation (Intermediate FU).

The *Output Valid If != 1* block is implemented using two FUs. The ALU of the (A) FU is configured to subtract the output of the *Barrel Shifter* from the *Right Input Register* and the constant 2 is programmed into the *Left Input Register*. In this way, when the decremented value reaches 1, a barrow will be generated from the carry out of the ALU, and this will propagate through to the *FNOut* flag to the *Select Start* and *Output Valid If FN* blocks The barrow out will cause the system to reset and the result to be sent out the data port. In the (B) FU of the *Output Valid If != 1* block, the *Conditional Unit* is set to bypass the ALU result and forward the *Right Input Register*. The *FN* flag that is generated by the (A) block is used to compute the valid bit for the output of the (B) block, thus performing the intended function.

The *Select Start* block simply uses the *Conditional Unit*, as controlled by the *FN* flag, to implement the multiplexing function between the constant 1 and the partial product produced by the multiplier. Likewise, the *Output Valid If FN* block sets the computation of the valid bit to depend on the *FN* flag and the valid bit of the *Left Input Register*. The ALU simply passes the result of the *Barrel Shifter* and the configuration is complete. Only when the value in the *Left Input Register* is valid, and a barrow is generated from the *Output Valid If != 1* block (A) ALU, will the output of this block be valid.

Both the *Select Valid* and *Hold Partial Sum* blocks supply values for the multiplier inputs and so they are placed at the bottom of the mesh. However, notice that the *Select Start* and *Output Valid If FN* blocks receive *Multiplier* outputs, but are not at the top of the mesh. The reason is that both of these blocks must receive the low word of the *Multiplier* output simultaneously. There are three ways to implement this type of arrangement: (1) connect the *Multiplier* directly to a *Skip Bus* input along the top of the mesh and use that to broadcast to both

FUs simultaneously, (2) connect the *Multiplier* to an FU along the top of the mesh through a local input and then use that FU to broadcast to the other two using the *Skip Bus* or (3) configure the crossbar for broadcast programming so that the output of the multiplier is sent to all eight mesh inputs simultaneously and pick two FUs along the top. All three of these are possible in this situation.

5.3.2.1 Broadcasting Using The Crossbar

The last method, configuring the crossbar for broadcast mode, is tricky to implement. This is because the programmer must make sure that part of the stream header is still passing through all crossbar connections leading up to the point of the broadcast at the time that the broadcast is issued. Because of the way that the crossbar is designed, once a given switch in the crossbar is configured, it will ignore all changes to that configuration until after the stream header ends. If this were not true, the broadcast configuration packet for the crossbar would reset the switch that was being used to route the stream from the data port to the crossbar and then to the mesh. The stream source would then be cut off, throwing off the entire process.

Fortunately, this possibility was foreseen, and the crossbar switches were designed to ignore all configuration changes until after configuration is complete. However, a given switch detects configuration completion by monitoring the stream's progress past that switch's point in the path. When the end of the stream header passes the switch, it enters data mode and is subject to reconfiguration again. Configuration may not actually be complete, though, and if the broadcast instruction is issued after the end of the stream header passes a switch, that switch will be reconfigured and the stream will be cut off. It is therefore the programmer's responsibility to

ensure that the stream header is still moving through all crossbar switches that could be adversely affected by the broadcast instruction at the time that it is issued. This can be accomplished by padding the stream header with NOP packets as described in the Colt detailed design Section A.1.5. In most cases, the stream header will be long enough that this would not be necessary.

5.3.2.2 Broadcasting Using An Intermediate FU

The second method of implementing the broadcast from the *Multiplier* to the *Select Start* and *Output Valid If FN* blocks is what is shown in Figure 5.11. There is only one connection from the *Multiplier* into a local connection at the top of the mesh. The FU at that point then broadcasts the unchanged result to the two FUs that need it. The result arrives at both simultaneously, satisfying that restraint. The total delay from the *Hold Partial Sum* FU through the *Multiplier* and back to the two blocks is then 5 clock cycles. Notice that it would only be four: crossbar (1), multiplier (2) and crossbar again (1), if not for the inclusion of the extra FU to perform the broadcast function. Three FUs are needed to provide the delay of 5 clock cycles required for the *Multiplier Delay* block so that the decremented operand arrives at the *Output Valid If != 1* block at the proper time. Note that if the intermediate FU had not been included at position A4 only two FUs would have been needed to provide the 4 clock cycle delay. The included FU does provide an easy means of establishing a configuration path through the structure; however, because configuration information can only be passed through local connections. The local connection point provided by the intermediate FU provides a convenient connection point from the multiplier for programming purposes. This becomes important for the reasons discussed in Section 5.3.2.3.

179
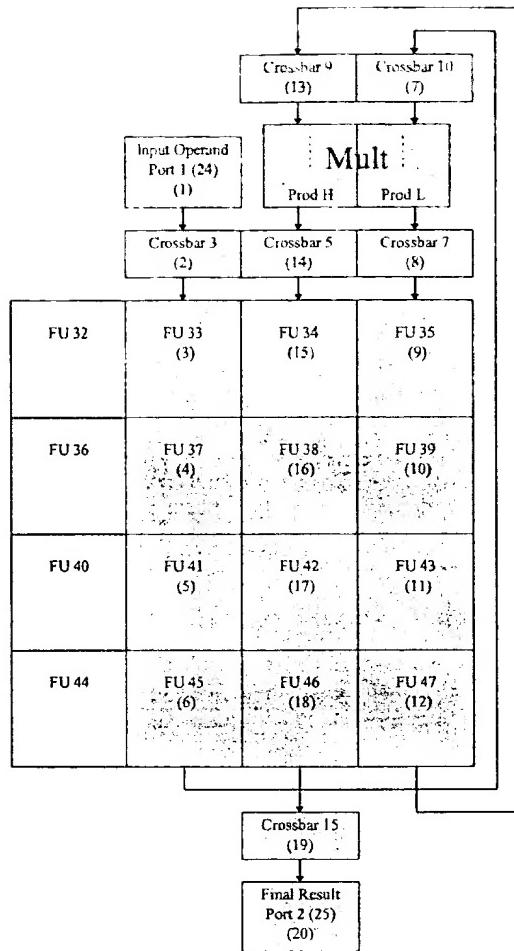
Figure 5.12 - Configuration Path (Intermediate FU).

The stream path used to configure the implementation shown in Figure 5.11 is given in

Figure 5.12. A single, non-branching stream has been used to configure the entire path through

the design. Notice that the input sources to the multiplier have been reversed from the original

conception of Figure 5.10. The reversal of the input sources allows the configuration to be

performed without using branching within the configuration stream. If the input sources were

not flipped, the low word of the multiplier output would have entered the mesh at the column

180

where the output port is connected. Once leaving the bottom of column 3, the stream must propagate to *Data Port 2*, meaning that it is not possible to loop back and program the eastern column. In order to program the eastern column, the stream would have to have split so that it could have propagated to the right and configure that column as well. Since it makes no difference which multiplier input the operands enter, the input sources were flipped.

5.3.2.3 Broadcasting Using The Skip Bus Directly
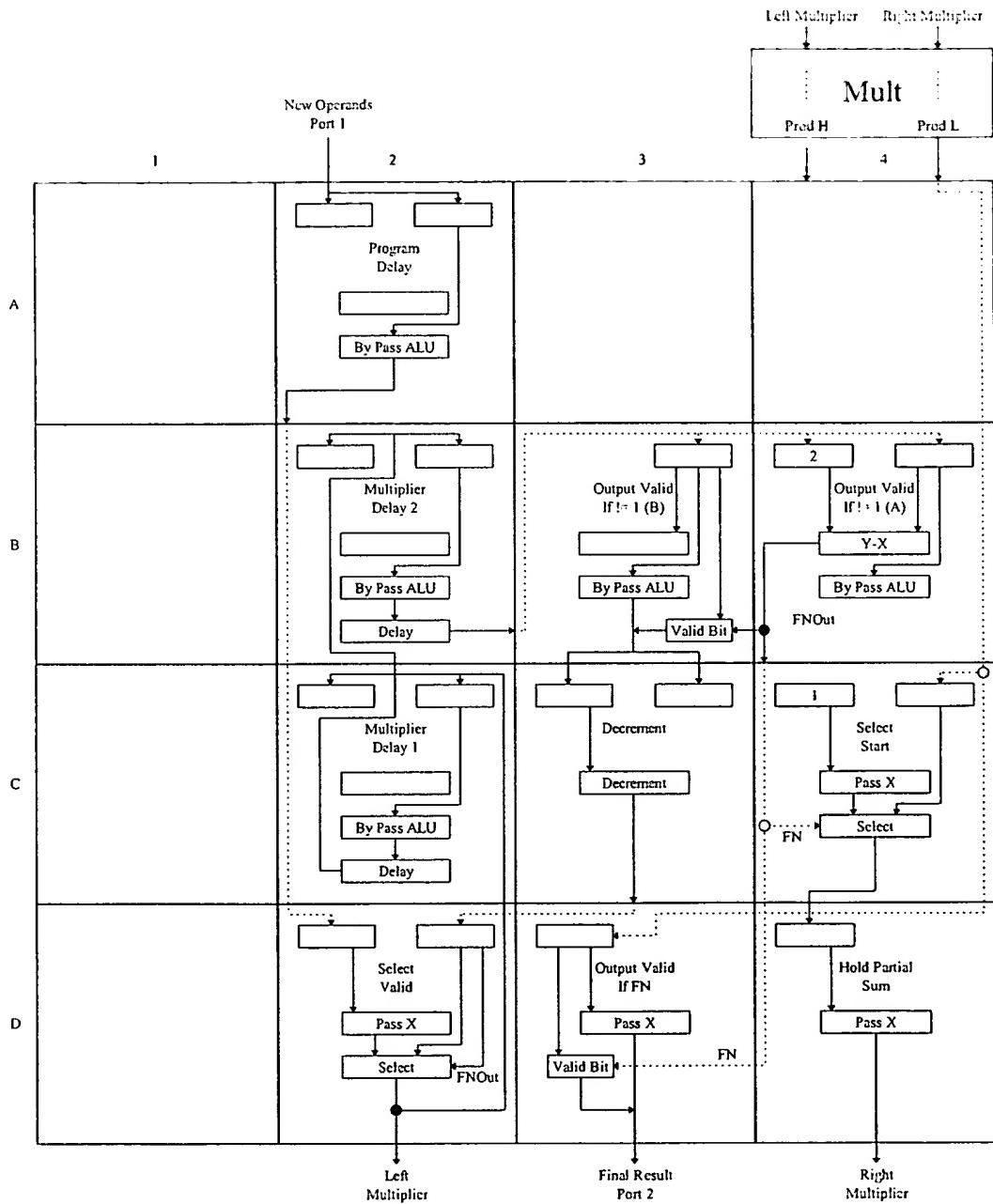
Figure 5.13 - Colt Factorial Implementation (Direct Skip Bus).

182

The third method of implementing the *Multiplier* broadcast is to connect the *Multiplier* output to one of the *Skip Bus* connections along the top of the mesh and use it to broadcast to both functional blocks simultaneously, as illustrated in Figure 5.13. That arrangement is difficult to establish for this particular function; however, because programming information cannot be transmitted through a *Skip Bus* connection. Since the stream header must end at the point where the crossbar connects to the *Skip Bus* along the top of the mesh, the Stream must effectively stop at that point. Since there is only one stream to work with in this design, entering through *Port 1* and exiting through *Port 2*, breaking the path in that manner would mean that a split programming path would be needed to continue the programming process. This can be a sticky situation, since the crossbar can only be programmed by the first packet that it receives. That means that the programmer has three choices when programming the crossbar: he can configure it in broadcast mode, he can configured it with a single new output, or, if the address of the first packet does not match any crossbar output, the output(s) that the crossbar was previously configured with can be used. So, if divergent stream paths are used, the programmer must be careful to ensure that the first time a branch of the stream gets to the crossbar, the leading packet programs it correctly. If the timing of the divergent stream is such that packets from one branch arrive at several different crossbar connections before the packets for programming those connections, the crossbar will be incorrectly configured.

This situation could arise, for example, if a stream diverges at an FU that lies along the bottom of the mesh. The FU at the point where the stream diverges will be configured to send programming information along two different routes, say south and east for the sake of argument. The FU programming scheme is such that it will begin forwarding all configuration information

in both directions as soon as it has removed its own configuration information from the stream header. Thus, an FU at the bottom of the mesh will begin forwarding configuration information to the crossbar and to the neighboring FU immediately after its own configuration. In order to properly configure the crossbar, the path leading to the crossbar must be configured before the path leading to the eastern FU for the reasons detailed above.
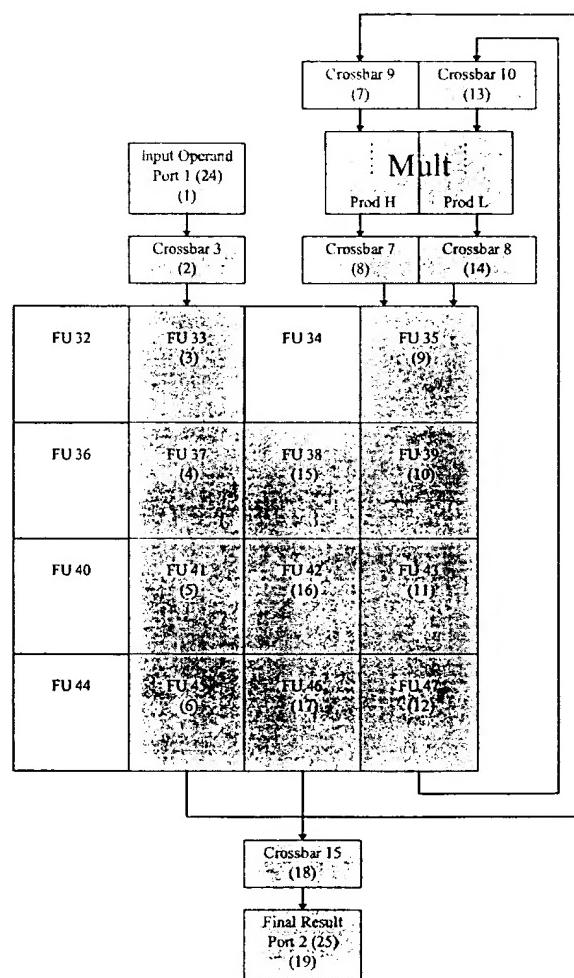
Figure 5.14 - Configuration Path (Direct Skip Bus).

Figure 5.14 shows the programming path used for the configuration given in Figure 5.13. Notice that the arrangement of left and right inputs to the multiplier agrees with the conceptual diagram and that FU 34 is now not programmed at all. This is possible because the stream header branches at FU 37. Branching here allows the remainder of the stream header to be forwarded in both the south and east directions. During programming, the packets in the stream header are arranged so that the path to the south is configured first, as indicated by the numbers in parenthesis. Column 2 is configured, followed by the left multiplier connection and then column 4 and back around to the *Skip Bus* input for column 4. The stream header will dead end there because configuration information cannot be forwarded through the Skip Bus. After the packet configuring Crossbar 8 passes through FU 37, succeeding packets will configure FUs 38, 42 and 46 and then proceed to the crossbar and out *Data Port 2*. The remainder of the stream header may actually exit the chip before the alternate path is fully programmed because of the latency of the pipeline that terminates at the *Skip Bus* connection of the fourth mesh column. In that sense then, the numbers in parenthesis indicate the ordering of the packets within the stream header, not necessarily the order of configuration.

By connecting the output of the multiplier directly to the *Skip Bus* in this way, two fewer FUs are needed for actual computation and could be programmed to augment the algorithm to allow for negative numbers, etc. This is one possible placement strategy and it may be possible to position the FUs so that a more compact section of the mesh is used for the function. Notice, however, that the first column of the mesh is entirely untouched, and can be freely used for other functions.

6. Experimental Results

Many of the merits of Wormhole RTR are well exemplified by the Colt device. There are also several additional features of Colt, unrelated to Wormhole RTR, that differentiate it from other FPGAs and add to its computational value. Comparisons of execution speed are difficult because the actual performance of the computation is heavily dependent on the implementation used, which is a function of programmer ability, simplifying assumptions offered by the problem, hardware capabilities and the nature of the computation itself. Perhaps for these reasons coupled with the youth of the field, there is little or no theoretical development for the performance of CCM systems in the literature. In order to better quantify the contribution of these improvements, an example will be described here that can be used as a basis for comparison. Since Colt was targeted toward DSP type applications, an example will be used from that problem domain.

6.1 CDMA - A DSP Processing Example

To give an understanding of the types of operations performed in a DSP application, the following extended example from communications has been included. It is important to note the types and quantities of various operations performed as well as the data dependencies between operands.

Code Division Multiple Access (CDMA) communications systems are becoming increasingly popular as replacements for more traditional Frequency Division Multiplexing (FDM) and Time Division Multiplexing (TDM) communications systems. This is largely because CDMA offers higher noise immunity and higher channel capacity over the other techniques. Unfortunately, the computational complexity of CDMA systems is also much higher than that of either FDM or TDM. The need for high speed computation in CDMA spurred much of the research presented here. Though CDMA was the driving force behind the resulting architecture and implementation manifested in the Colt ASIC, the knowledge derived from this endeavor can be applied to the more general Digital Signal Processing (DSP) problem and even to general purpose computation.
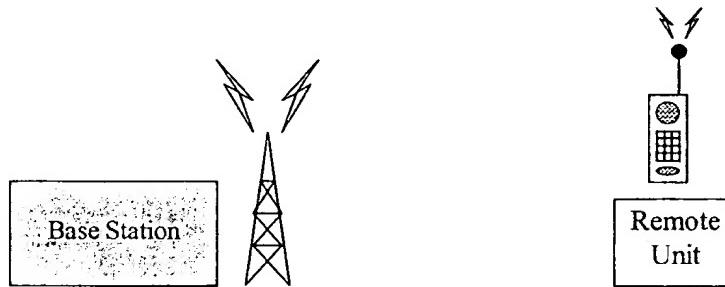
Figure 6.1 - System Concept.

There are many methods and algorithms for implementing a CDMA class communications system [72][73]. For example, the research conducted in the Global Mobile Radio (Glomo) project at Virginia Tech applies the concept to cellular phone systems, such as

---

72 Liberti, J. C. and Rappaport, T. S., "Capacity Improvements in CDMA Cellular Systems," *IEEE Transactions on Vehicular Technology*, vol. 43, no. 3, August 1994.

the one pictured in Figure 6.1. Seen here are the components for a single cell of a phone network. There are many such cells set up in a honeycomb pattern across the nation to give seamless wireless phone service to customers everywhere. Within each cell, there are many simultaneous users that each require their own communication channel. These are represented by multiple remote units in the form of telephone handsets. As a given user moves from region to region, he also moves from cell to cell, forcing the hardware to dynamically reconfigure the communications scheme, shifting to an available channel at the new base station in the new cell. Normally this reconfiguration is done transparently to the user so that no intervention is required and no interruption of service is perceived.
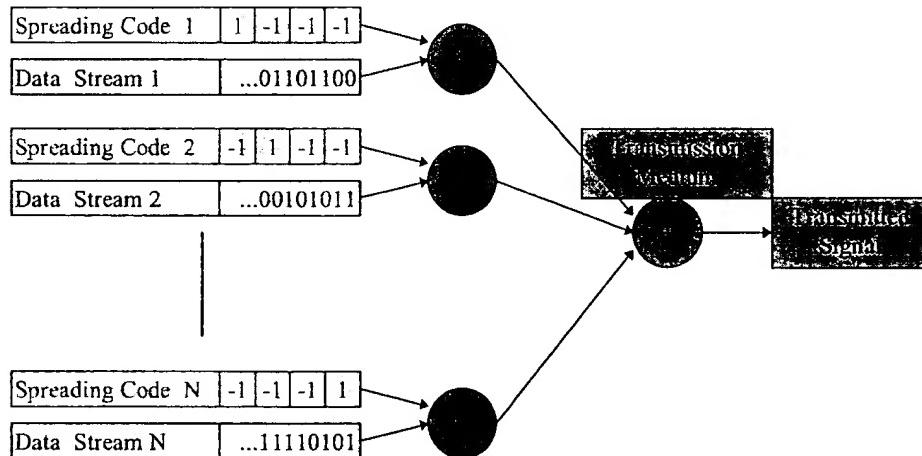
Figure 6.2 - CDMA Transmission.

---

73 Liberti, J. C., "Analysis of CDMA Cellular Communication Systems Employing Adaptive Antennas," Ph.D. Dissertation, Bradley Department of Electrical Engineering, Virginia Tech, August 1995.

Whereas TDM systems simply require a temporal shift in the reception of packets of information and FDM systems merely require switching to a new frequency, CDMA systems require more radical changes in the transmission and reception subsystems to multiplex channels. The reason why is intimately related to the principles of CDMA. The transmission side of a CDMA system is shown in Figure 6.2, which shows several users simultaneously transmitting data on N channels. Both a data stream and a spreading code are associated with each user. In a CDMA system, all users transmit simultaneously across the entire communications spectrum, and the spreading code is used as a key to separate channels. The spreading code is itself a fixed length bit sequence the length of which is specified as a system parameter. Each bit of the data stream is multiplied by the entire spreading code vector, forming a new vector the same length as the spreading code. This new vector is then transmitted sequentially through the communications medium (a 0 in the data stream is interpreted as a -1 for purposes of the multiplication). In the transmission medium, all users' encoded transmissions overlap in both the spectral and temporal domains because they are using the same frequencies at the same time. The spreading code vectors are chosen to be as orthogonal as possible to one another so that any two transmissions being sent at the same time using different spreading codes will additively cancel each other in the transmission medium. Thus, the summation of all the transmissions effectively appears as low level noise in the medium.
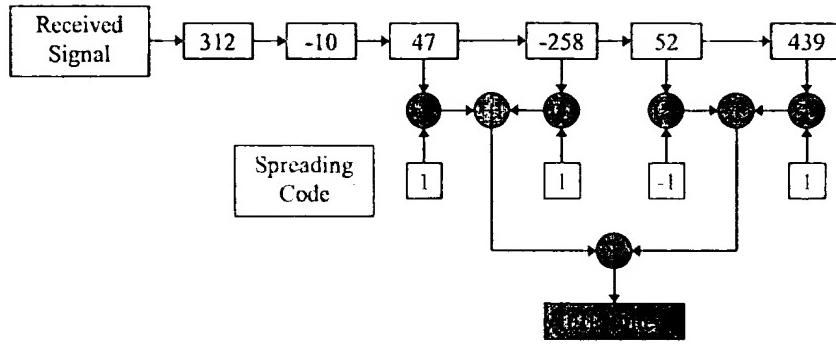
Figure 6.3 - Matched (FIR) Filter.

At the reception end of the system, shown in Figure 6.3, the signal received (appearing almost as noise) from the transmission medium must be decoded into one of the original data streams. To accomplish this, the receiver is configured with the spreading code of the desired data stream. Then, the signal received from the transmission medium is digitized and passed through a matched filter, typically implemented as a Finite Impulse Response (FIR) filter that has been pre-configured with the spreading code of the desired signal. The matched filter functions as a template matching mechanism similar to that in a neural network. In general, the weights used in the filter are arbitrary values that can be calculated in many ways, including simulated annealing [74]. For purposes of this application, the weights are either +1 or -1. In Figure 6.3, a four bit spreading code is being used; requiring a four tap FIR. Each sample of the digitized signal passes through the four shift register stages. At each stage, the signal sample is multiplied with the spreading code bit corresponding to that position. These products are then summed to form the output of the FIR for that clock cycle. The shift stages are then advanced, taking in a

190

new signal sample on the left hand side and the process is repeated. The output of the FIR will spike when the transmitted vector in the signal stream is aligned with the spreading code vector. If the output spike is positive, then a 1 was being transmitted, if the output spike is negative, then a -1 (or 0) was being transmitted. This is a simplified view of the actual operation of the system and it avoids many issues, but it is representative of the essence of the process.
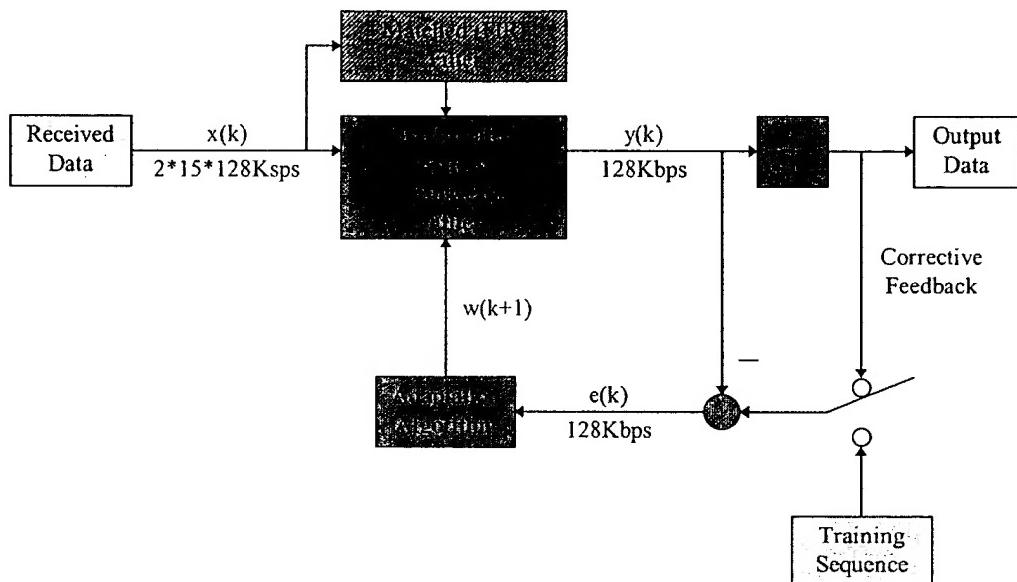
Figure 6.4 - Mobile Unit Receiver Architecture.

A generic architecture for the receiver of the remote unit of the system is shown in Figure 6.4. There are two stages in the operation of the receiver. First, during the acquisition phase the FIR Filter is used to "lock on" to the incoming data signal so that the received signal is aligned with the spreading code vector being used. This synchronization information is fed to the

74 Nobakht, Ramin A., "Optimization of IIR Filter Coefficients from FIR Filter Taps by Mean Field Annealing," *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. v-77 - v-80, 1992.

Adaptive Filter, which is represented by the *Fractionally Spaced Transversal Filter* and the *Adaptation Algorithm* in the figure. Once initial acquisition has been completed the FIR filter is no longer used. During the operational phase of the receiver, the *Adaptive Filter* is used to perform fine grain tracking of the incoming signal and to dynamically update the weight vector $w(k+1)$ to allow for variations in the received signal due to noise and various types of interference. The end product of the *Adaptive Filter* is a sampling of the transmitted bit stream $y(k)$ that is produced at the data rate of 128K samples per second (bps). Each sample in this stream is then sent through a threshold operator to recover the transmitted bit stream at the rate of 128K bits per second (bps).

From this architecture, an order of magnitude for the number of computations required can be deduced. In a typical system [5], the data rate is 128Kbps, with a spreading code vector length of 15 chips and two samples being taken per chip. Hence, the number of samples per second coming into the system, shown on the left of the figure, is 2*15*128K = 3.93 million samples per second (sps). This represents the number of evaluations per second of the FIR Filter during the acquisition phase of mobile receiver operation. However, this is not the worst case of operation for the entire system. In the base station, similar architectures are found, but in that case four samples per chip are required. This results in a total of 4*15*128K = 7.86 million sps. Since four samples are taken per chip and there are 15 chips in the spreading code vector, a 60 tap FIR Filter must be used in the base station. An adder tree for a 60-tap FIR Filter requires 30+15+7+4+2+1 = 59 adders. Hence, the operation count for the FIR Filter is 59*7.86 million = 464 million additions per second.

It should be noted that the real-time requirements of the communications system further specify that a maximum latency of no more than 10 ms can be tolerated in the processing of the antenna data. Further, the data samples arriving from the antenna are 12 bits wide. It will be assumed that the width of the adders can be limited to 16 bits in width without risk of overflow due to additive cancellation of the transmitted chips.

6.2 Relative Performance

Four different chips will be compared: Colt, Xilinx XC4013E, Xilinx XC6216 and a generic microprocessor. The two Xilinx parts are believed to be representative of the state of the art in FPGA design. They are similar in functionality, speed and design to other devices on the market. The generic microprocessor specifications will be taken from the derivation given in Section 4.1.

6.2.1 Performance Metrics

The evaluation of the 60-tap FIR filter used in the base station receiver will be used for purposes of comparison. It is believed that this function places the highest demands on the computational engine and so will establish worst case performance. None of the single chip platforms discussed can achieve the performance that this application requires. In order to facilitate comparison between these systems, a standard of comparison will be established based on the number of evaluations of the 60-tap FIR filter that can be performed within the 10 ms window of acceptable latency. The requirements of the system stipulate that the filter must be evaluated 7.86 million times per second, or 78,644 times per 10 ms window. The significance of

the number of evaluations per 10 ms window will become apparent as the effects of run-time reconfiguration are taken into account.

The equation representing the number of evaluations per window is:

$$N_{eval} = \frac{T_{win} - T_{setup}}{T_{exec}}$$

where, N_{eval} is the number of evaluations per window, T_{win} is the width of the window (10 ms), T_{setup} is the total time required configure the system to perform the computation and T_{exec} is time spent computing. The effects of pipeline initialization and termination will be neglected since these are expected to be less than six clock cycles per reconfiguration, which is insignificant next to the size of the 10 ms real-time window. The T_{setup} parameter accounts for the run-time reconfiguration time of CCM implementations and will be zero for microprocessor based solutions. The T_{exec} parameter will include instruction fetch, branch miss and other associated overhead for microprocessor systems as well as the time of instruction execution. For CCM implementations, T_{exec} will be the time spent in instruction execution.

Finally, it is unfair to compare the performance of various implementations without taking into account the amount of silicon that is required by each. Typically, this information is not available to the general public for commercial devices. In cases where it could be obtained, the die size and process size will be used to normalize the results to use units of λ^2, as discussed in Section 2.1.2. It should be noted that there are many factors beyond computation style that affect the die area required for a particular computation. These include process resolution specifications as well as layout effort and experience. When comparing to other

194

implementations. it is important to keep in mind that Colt was made using a small, inexperienced university design team using a process that is not state of the art.

6.2.2 FIR Implementations

There are three classes of computational engines being compared: Wormhole RTR driven processors (Colt and Stallion), conventional FPGAs (Xilinx XC4013E and XC6216) and a normal control flow processor. Each requires a slightly different implementation strategy. These strategies will be outlined here so that the conditions of comparison can be fairly judged.

The microprocessor implementation is the most conventional, involving a simple sequential execution of all the operations. From the derivation in Section 4.1, $T_{exec} = 2.52$ for an integer-only processor in which it is assumed that the addition operation itself can be performed in a single clock cycle. There are 59 additions/subtractions per evaluation of the FIR filter, for a total of 148.68 clock cycles per evaluation. Assuming a clock rate of 200 MHz, the number of evaluations per 10 ms window is then $N_{eval} = 200$ MHz / 148.68 * 10 ms = 13,451. For a one second window, N_{eval} climbs to 1.34 million evaluations per second.
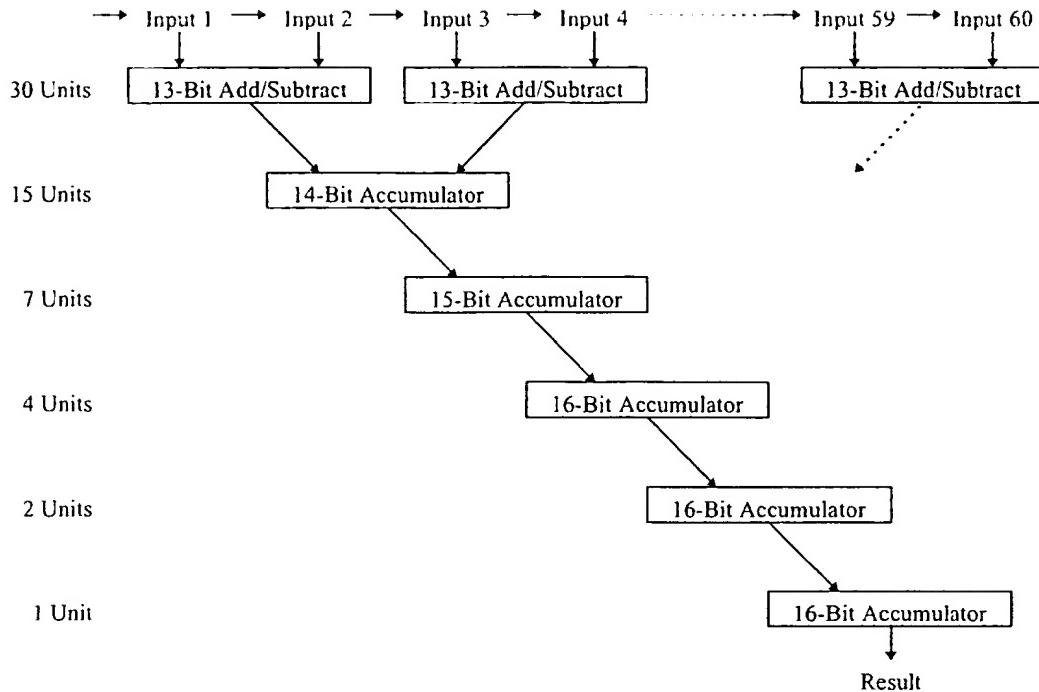
Figure 6.5 - CCM Adder Tree Implementation.

The implementation of the FIR filter on the Xilinx parts requires run-time reconfiguration because the entire filter cannot be statically configured onto either chip. However, in these devices the luxury of a bit-oriented architecture can be exploited to tailor the widths of the adders to the precision required at a given level in the adder tree. A tree of adders will be constructed within these devices to form a pipelined structure as shown in Figure 6.5.

The XC4013E device contains a 24x24 array of 576 CLBs. The number of CLBs required per adder is $n/2 + 1$, where n is the bit width of the adder. Also, a number of CLBs must be used as routing resources to implement the tree structure. From experience using these devices, it seems reasonable to assume that 25% of the CLBs used for the entire implementation will need to be sacrificed for routing.

196

Table 6.1 - Xilinx XC4013E FIR Adder Resource Requirements.

Level	Number of Adders	Adder Width	Number of CLBs/Adder	Total CLBs	Total CLBs With Routing
1	30	13-Bit	8	240	320
2	15	14-Bit	8	120	160
3	7	15-Bit	9	56	75
4	4	16-Bit	9	36	48
5	2	16-Bit	9	18	24
6	1	16-Bit	9	9	12
				Total	639

In addition to these resources, CLBs need to be used to create the 12-bit wide delay stages for the data being shifted across the top of the mesh. There are two flip flops per CLB, requiring 6 CLBs per stage for 60 stages, giving 360 CLBs for this purpose. Normalizing for routing area required to deliver the data to the first level adders gives a total of 480 CLBs to implement the delay stages.
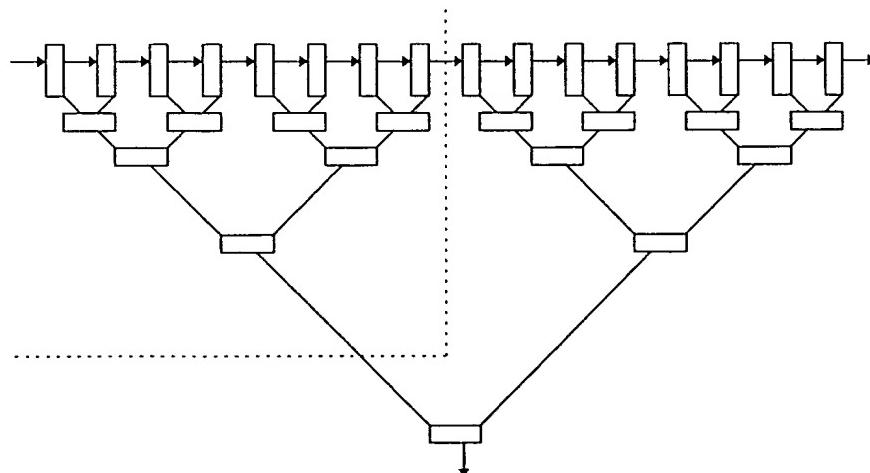
Figure 6.6 - Xilinx XC4013E & XC6216 FIR Filter Partitioning.

One way to implement this function on the XC4013E is depicted Figure 6.6. Although a smaller version of the full tree is shown here, this is the method used in a generic sense. The

197

dotted line indicates the separation between the two partitions of the design. The data is streamed into the left partition and then out the other side so that it can be buffered and streamed into the right partition. A new partial result is produced on each clock cycle, and these are buffered as well. When the right partition is executed, these are injected using the appropriate delay. With the resources split in half for all but the last level, 320 CLBs are required per partition.

According to Xilinx [11], an 8-tap FIR filter of this design using 8-bit data can run at 10 MHz. A 16-bit adder is rated to run at 76 MHz on the XC4013-2 architecture and so most of the delay is due to routing constraints. Likewise, it will be assumed that a 30-tap FIR filter can be implemented as an extension of a 8-tap filter. Thus, the operating rate of 10 MHz will be taken as an upper bound on the clock rate of this implementation. The XC4013 requires 247,960 configuration bits which are serially clocked in at a maximum rate of 10 MHz; thus, configuration of the device requires 24.8 ms.

Using these parameters, N_{eval} can be determined. Two configurations are required, giving a value of 49.6 ms for T_{setup}. Two clock cycles are required for each result since both partitions must be executed, producing a result every $T_{exec} = 200$ ns. Because the value for T_{setup} exceeds the T_{win} parameter of 10 ms, it is clear that a single XC4013 is not capable of successfully implementing the FIR filter in real-time. As a point of comparison, if the acceptable window of delay were extended to one second, N_{eval} would be 4.75 million evaluations per second.

Table 6.2 - Xilinx XC6216 FIR Adder Resource Requirements.

Level	Number of Adders	Adder Width	Number of Cells/Adder	Total Cells	Total Cells With Routing
1	30	13-Bit	52	1560	2080
2	15	14-Bit	56	840	1,120
3	7	15-Bit	60	420	560
4	4	16-Bit	66	264	352
5	2	16-Bit	66	132	176
6	1	16-Bit	66	66	88
				Total	4,376

The derivation for the Xilinx XC6216 is similar, with the notable difference that four cells are required to implement a single bit of an adder and only a single flip flop is contained per cell. Table 6.2 shows the number of cells required for the adder tree. In addition, 60 12-bit delay stages must be implemented, requiring an additional 60 * 12 = 720 cells, plus 25% routing overhead, giving a total of 960 cells for the delay stages. The XC6216 contains 4,096 cells, which is insufficient to meet the requirement of 5,336 cells for the filter. The same partitioning may be used as was used for the XC4013.

As of this writing, the XC6216 is not a released product and the preliminary Xilinx data sheets do not list any example operating frequencies. However, it has been the experience of our research team that designs on the XC6216 generally operate slower than on the XC4000 series parts. Xilinx recommends using the greater number of flip flops on the XC6200 series to further pipeline operations in order to gain increased speed; however, this would not be a fair comparison since this was not done in the other implementations. Using this knowledge, it seems reasonable to use the same upper bound of 10 MHz for the XC6216 operating frequency as was used for the XC4013E.

199

The XC6216 can be partially configured so that only configuration bits that are contributing to the computation need to be written onto the chip. Further, the XC6216 supports a broadcast programming mode in which multiple identical cells can be programmed simultaneously. It will be assumed that the delay buffers can be arranged so that they can all be configured using a single configuration write cycle. Likewise, using broadcast programming mode, it will be assumed that each adder can be configured using four configuration write cycles. For each configuration write cycle, two clock cycles must be used to set the column broadcast mask and then to write the configuration data. Each column of the XC6216 is 64 cells tall. Since the units used for the FIR filter tree are at most 16 cells wide, the row decoder address should only need to be set every fourth configuration write cycle, giving ¼ of a clock cycle for this write. Thus, it will require a total of 2.25 clock cycles to perform a single configuration write cycle. For the left partition, there are $15 + 8 + 4 + 2 + 1 = 30$ adders and 30 buffers in the adder tree. The XC6216 supports broadcast programming so that each of the adders may be configured using a single broadcast write for each of the four sets of common cells in the entire width of the adder. The buffers could perhaps all be configured using a single configuration write cycle plus 25% routing area, giving 1.33 configuration write cycles to configure the buffers. There are four groups of cells for each adder that could be programmed using broadcast mode. Allowing for 25% routing in the design, this results in 5.3 configuration write cycles per adder. In total, $(1.33 + 5.3 * 30) * 2.25 = 361$ 40 ns clock cycles would be required to configure the left partition of the adder tree for a time of 14.4 μs.

The right partition can take advantage of the partial configuration abilities of the XC6216 by using essentially the same adder tree and only modifying the adders/subtracters along the first

level to match the filter weights at those positions. 15 such units of 4 cells each would need to be reconfigured for a total of 4 * 15 = 60 broadcast configuration write cycles that may be required. This would require 2.25 * 60 = 135 40 ns clock cycles for a right partition reconfiguration time of 5.4 µs.

Using these parameters, N_{eval} can be determined for the XC6216. The total configuration time is T_{setup} = 19.8 µs. Two clock cycles are required for each result since both partitions must be executed, producing a result every T_{exec} = 200 ns. N_{eval} for the 10 ms window is 49,901. For a one second window of acceptable delay, N_{eval} would be 4.99 million evaluations per second.
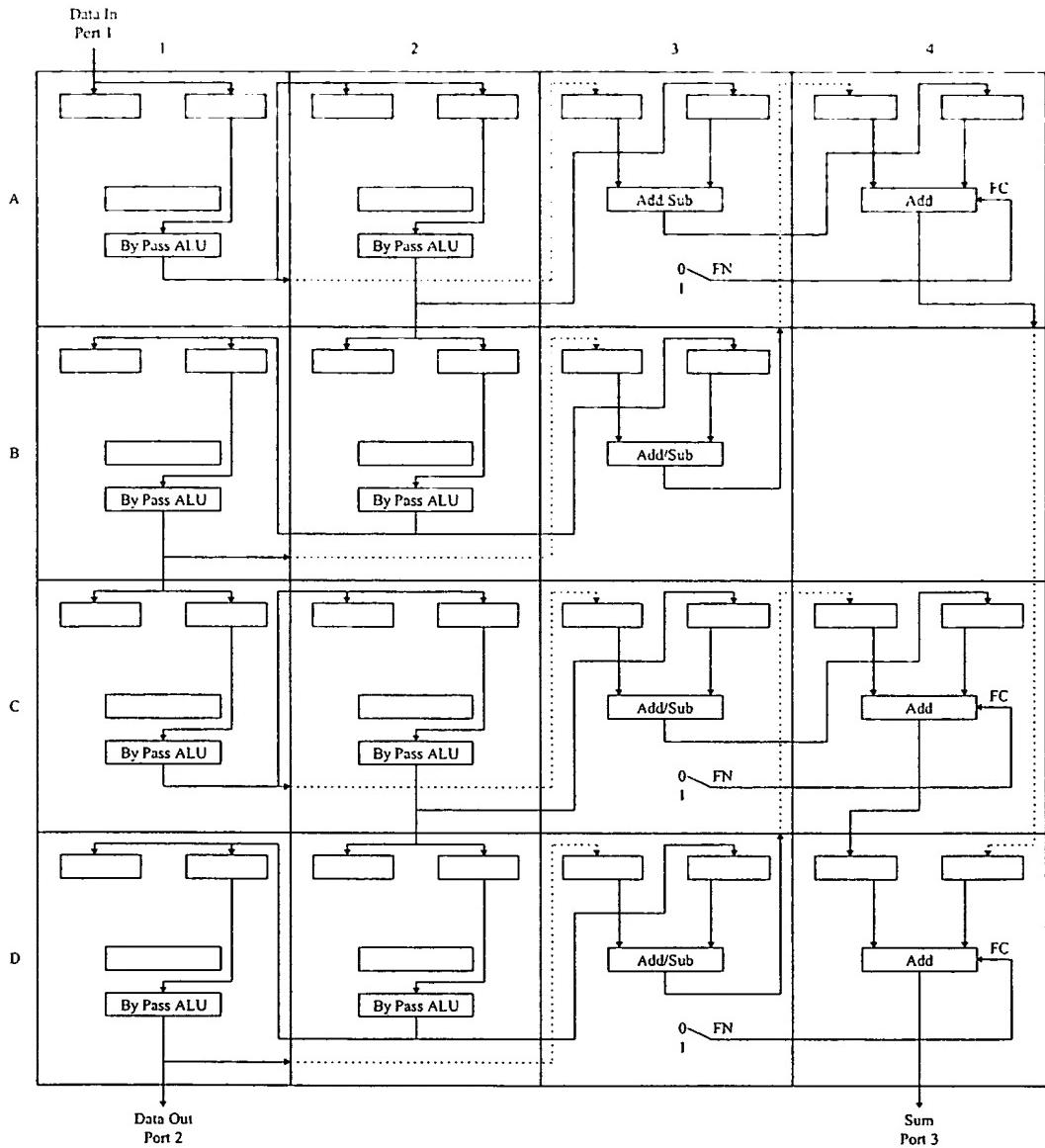
Figure 6.7 - 8:1 FIR Filter Implementation on Colt.

The implementation of the FIR filter on the Colt requires more reconfigurations because there are fewer resources available on the chip; owing to its smaller physical die size. Resource constraints dictate that the maximum size FIR filter that can be implemented on Colt is 8:1, as

shown in Figure 6.7. If a "smart" external memory controller, such as a Stream Controller, is assumed, this number could be higher; however, this was not considered to be fair for purposes of this comparison. With an 8:1 FIR filter the sample data stream can flow through the chip and out a data port, just as is required for the Xilinx implementations. The FUs in column three implement the first level adders for the FIR filter. Note that depending on the weights required by the filter, these FUs will need to implement one of the functions $X+Y$, $X-Y$, $-X+Y$ or $-X-Y$. The first three functions can be implemented using the single ALU contained within each FU. The function $-X-Y$ cannot be implemented using a single ALU. To implement the function, both input operands must be inverted, added together, and then the result must be incremented twice. The result may be incremented once by setting the carry in of the first level adder to 1. The switches on the FN flag shown in FUs A3 and C3 allow the same FU to control the carry in for the second level adder, which can increment the result a second time. The switch on the FN flag from FU D4 allows an optional increment of the result for cases in which several of the first level adders must implement the $-X-Y$ function. For example, if the four weights in the left hand tree were all -1, then the two first level adders could implement the function $X+Y$, and the second level adder could be programmed to output an inverted result which the third level adder would then increment. If all eight weights are -1 it may be necessary to use FU B4 as part of the tree to perform a final negation, but this case is considered to be rare.
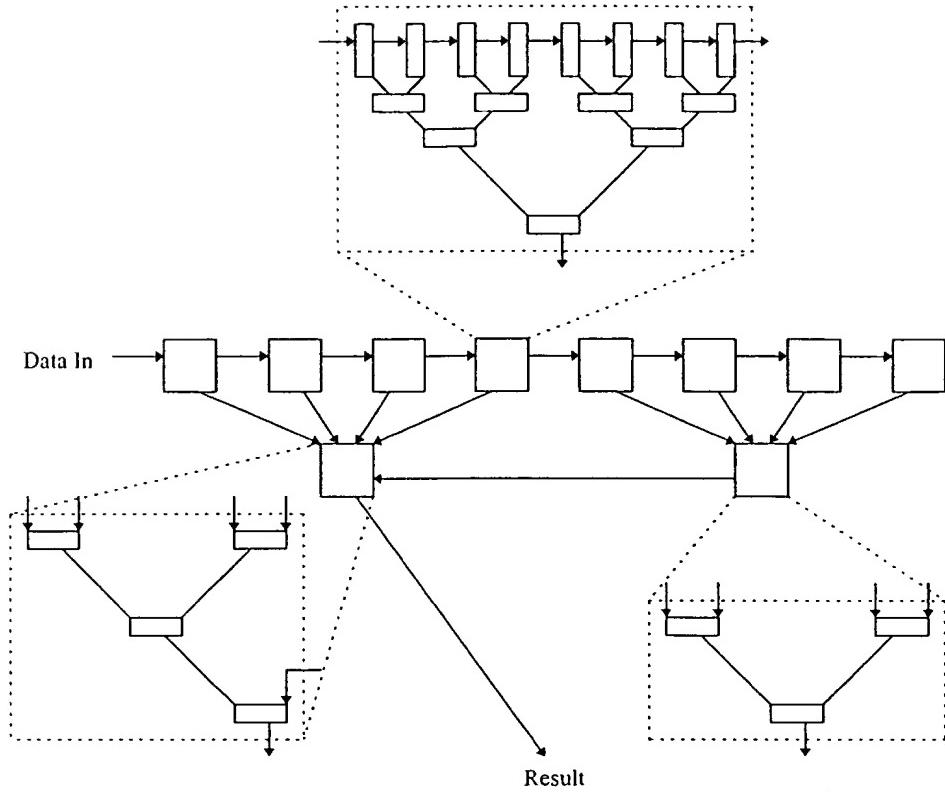
Figure 6.8 - Colt FIR Filter Partitioning.

Figure 6.8 shows the flow of data between configurations. The data shifts through a row

of eight different configurations. Each configuration is an 8:1 FIR filter tailored to the weights

used at the eight delay stages used at that point in the overall 60-tap FIR. An enlargement of one

of these eight stages is shown above the input row in Figure 6.8. Two additional configurations

collect the partial sums from the input row and produce the final result. Note that the collection

stage on the right feeds the final partial sum from the right half of the tree into the left collection

stage so that the final sum can be computed. The left collection stage is I/O bound since all six

data ports are used to accept partial sums from the various other stages.

The operating frequency of Colt is at least 50 MHz, giving a maximum clock period of 20 ns. Since 10 configurations must be run in order to produce a result we have a value of T_{exec} = 200 ns. T_{setup} can be determined by summing the configuration times for all 10 stages. The configurations for each of the input row stages will be largely the same except for the initial row of adder/subtracters. The configuration of these stages is hindered somewhat by the fact that there is only one input port, and thus only one port is used to inject configuration information into the chip. In the first of these, all 16 IFUs must be configured as well as two crossbar configurations and three data ports, for a total of $16 * 8 + 2 + 3 = 133$ clock pulses spent in configuring the chip. The following seven can take advantage of Single Word Program mode in order to skip over the existing configured IFUs and only modify the four performing the first row of adds/subtracts. The path for this would proceed through a data port, a crossbar connection, using Single Word Programming on two of the IFUs and then fully program four more, for a total of $1 + 1 + 2 + 4 * 8 = 36$ clock pulses. The worst case configuration path through the left collection stage passes through two data ports, two crossbar connections and 4 IFUs for a total of $4 * 16 + 2 + 2 = 68$ clock pulses. The worst case path for the right collection stage is the same. For both collection stages, the rest of the configuration can be done in parallel with the worst case. This gives a total of $133 + 7 * 36 + 2 * 68 = 521$ 20 ns clock cycles spent in configuration for a T_{setup} time of 10.42 µs. N_{eval} is 49,947 per 10 ms window on Colt, with the 1 second window performance being 4.99 million evaluations per second.

Speculating on the performance of Stallion, Colt's successor, is difficult because a number of changes are planned that will improvement its standing. However, it is useful to do a simple extrapolation based on the Colt architecture. At least four times as many on-chip

205

resources are planned for Stallion in comparison to Colt, using an 8x8 mesh of IFUs. A total of twelve data ports will be supported, and the speed of the chip is expected to double to 100 MHz. Four times the die area can also be expected. The same partitioning used for the XC4013E could then be used for Stallion since it would contain 64 IFUs, which should easily be able to implement a 32:1 FIR filter. The configuration path through this device for the left partition would traverse two data ports, two crossbar connections and 64 IFUs for a total of 64 * 8 + 2 + 2 = 516 10 ns clock pulses giving a time of 5.16 μs. The right partition could partially reconfigure the first level adders using Single Word Programming to pass through to the needed devices. This path would pass through a data port, a crossbar connection, 4 IFUs using Single Word Programming mode and then an additional 15 IFUs would need to be fully configured. This process would require 1 + 1 + 4 + 15 * 8 = 126 10 ns clock pulses for an additional configuration time of 1.26 μs.

The total T_{setup} time would then be 6.42 μs. There are two pipelined configurations, each of which must execute once to produce a result, giving a T_{exec} time of 20 ns per result using a 100 MHz clock. The number of evaluations possible per 10 ms window then becomes 499,679, and the number of evaluations possible per one second window becomes 49.97 million.

206

Table 6.3 - FIR Evalutation Performance By Device.

Device	# Of Config.	T_{setup}	T_{exec}	N_{eval} Per 10 ms Window	N_{eval} Per 1 sec Window (Millions)
200 MHz RISC Microprocessor	N/A	0	745 ns	13,451	1.34
Xilinx XC4013E	2	49.6 ms	200 ns	Impossible	4.75
Xilinx XC6216	2	19.8 μs	200 ns	49,901	4.99
Colt	10	10.42 μs	200 ns	49,947	4.99
Stallion	2	6.42 μs	20 ns	499,679	49.97
			Target	78,644	7.86

While the area normalization is less difficult to compute, the availability of die and process sizes makes it more elusive. Fortunately, these parameters were obtained for a number of devices and could be extrapolated in a few additional instances. The microprocessor area is easily obtained, as Flynn defined the unit A to be equal to 4 million λ^2. The integer processor that he derives requires 40.05A for a total of 160.2 million λ^2. The specifications for the Xilinx XC4013E were obtained, and the area required is 85 mm^2/(0.3 μm)^2 = 944 million λ^2. This assumes that the 0.5 μm process used for the XC4013E has an effective λ of 0.3 μm, as is the case for Colt. Again, this is generous because if the process actually has an effective λ of 0.25 μm, as should be the case, the area consumed rises dramatically. Colt consumes 34.16 mm and it has an effective λ of 0.3 μm in a 0.5 μm process, giving it an area of 382 million λ^2. Stallion is given an area four times this figure. Table 6.4 shows the relative performance of the various devices normalized with respect to the area consumed by Colt.

207

Table 6.4 - Area Normalized FIR Evalutation Performance.

Device	λ (μm)	Die Area (mm^2)	Die Area (Million λ^2)	Eval. Per 10 ms Window	Eval. Per 1 sec Window
200 MHz RISC Microprocessor			160	32,115	3.21
Xilinx XC4013E	0.30	85	944	Impossible	1.92
Xilinx XC6216	0.30	100	1,111	17,157	1.71
Colt	0.30	34.16	382	49,947	4.99
Stallion	0.30	136.64	1,528	124,919	12.49
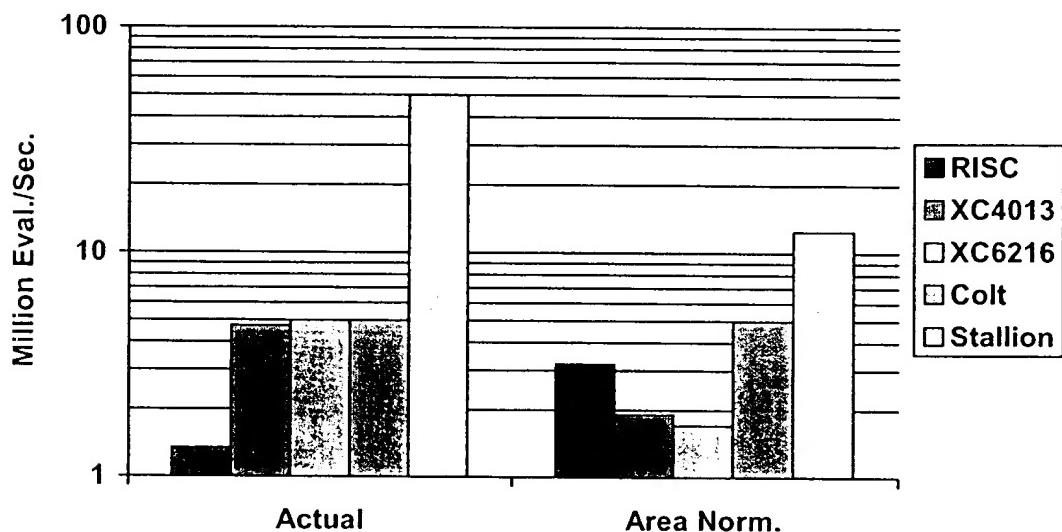
Figure 6.9 - Actual and Area Normalized FIR Evaluation Performance.

6.3 Colt vs. Microprocessors

The area normalized margin of Colt over the RISC microprocessor is not that great. As was previously mentioned, while area normalization is important to use as a basis for comparison, it must be noted that Colt was not designed by professionals using professional tools

and an aggressive VLSI process. It is, therefore, significant that there is any margin of gain over the other devices when comparing based on area normalized criteria. Further, it is not possible to fully account for the scaling factors involved in the speedup difference between devices because there are to many non-deterministic variables involved. For these reasons, the normalized area speedups are shown, but will not be justified. They are included here only to dispute arguments that Colt uses far more silicon than other implementations.

There are several factors, unrelated to area, that contribute to speedup factors and can be computed. In order to fully understand the sources of speedup and the various effects of the design decisions made on Colt, these will be outlined here.

6.3.1 Computational Density

Table 6.5 - Colt Unit Layout Sizes.

Unit	Configuration Storage Area (λ^2)	Total Area (λ^2)
Cross Bar Element	0	156,750
Data Port	23,400	1,035,400
Multiplier	0	4,400,000
Functional Unit (FU)	511,650	2,169,750
Interconnected Functional Unit (IFU)	511,650	11,440,000

The area utilization of each unit of the Colt chip is summarized in Table 6.5. There are six data ports, one multiplier, sixteen Interconnected Functional Units and 12*16-6*6 = 156 cross bar elements. Thus, the total area of the chip used by units is 218,105,400 λ^2. The chip was designed to be 5 mm on a side. A square chip 4.43 mm on a side is required in a 0.5 μm

209

(λ=0.3μm) process using these units. Of course, this does not include any area for global routing, clock, power, ground and inter-unit wiring. As a point of interest, the final die implementation of the Colt, including pads, global routing, etc, was 20,385 λ x 18,719 λ = 381,586,815 λ^2. The resulting die is 6.1 mm x 5.6 mm given the 0.5 μm (λ = 0.3 μm) process that was used.
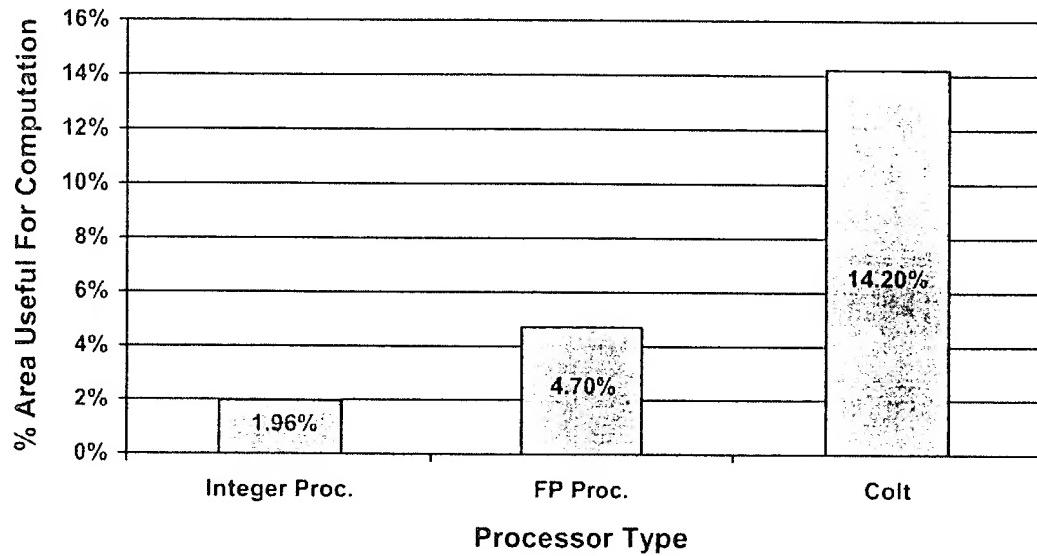
Figure 6.10 - Computational Density Comparison.

The area of the Colt that is used for computation can be taken as the area of the multiplier plus the area used by the sixteen FUs minus the area for the FU configuration information. The computational area of the chip is 30,929,600 λ^2, which means that 14.2% of the chip area is actively involved in computing results. This is over three times the estimate of 4.7% for a

210

microprocessor with a floating point unit and more than seven times the 1.96% estimate for a purely integer processor as described in Section 2.1.2. These results are summarized in Figure 6.10. This translates to more functional units per unit area that can be used to perform useful computational tasks in parallel. For applications in which this advantage can be exploited, such as the FIR filter, a speedup can be realized. As was noted in the introduction, area related advantages such as this are difficult to quantify in terms of a deterministic number and this one is noted here only as a point of interest.

6.3.2 Execution Overhead

As discussed in Section 4.1, an integer based microprocessor incurs a 1.52 clock cycle instruction execution penalty for every "computational" instruction that is executed. Over time this penalty will be seen as lost computational cycles. For a CCM device using a static configuration, such as Colt, the overhead of configuration is incurred once, and after that time every clock cycle can be devoted to computation. Thus, as a data run becomes large, the overhead of configuration becomes increasingly negligible, whereas a microprocessor's overhead remains a constant percentage of execution time. In the FIR example, Colt is spending 10.42 μs/10 ms = 0.1042% of the total execution time performing reconfiguration operations. The RISC microprocessor spends 1.52/2.52 = 60.3175% of the 10 ms window performing overhead operations. Naturally, this is in part due to the assumption that the operation being performed is addition, which only requires a single clock pulse to execute. If computational instructions are assumed to require more than a single clock pulse, the proportion of time spent in executing overhead will drop. However, the trend in DSP design seems to be toward using more and more

silicon to reduce the execution time of instructions. The Analog Devices SHARC DSP, for example, can execute floating point addition and multiplication in a single clock pulse. With these trends, the execution overhead of the microprocessor will become the limiting factor in control flow execution.

Of course, this discussion neglects the effects of run-time reconfiguration. In cases where short runs of data must be processed, Colt would not fair as well. If, for example, the FIR filter only needed to be executed once in the example above, Colt would require 10.42 μs to reconfigure and an additional 93 clock pulses to run the data through the pipeline for a total of 12.3 μs to produce a single result. The microprocessor would only require 745 ns to produce the same result. In all likelihood, the same type of configuration would not be used with Colt for this situation and a better method could be devised. Nonetheless, the point is well made that this type of processing does require large numbers of repetitive calculations before it becomes efficient.

The total advantage of Colt over the microprocessor due to execution overhead for this application then becomes (1 - 0.1042%) / (1 - 60.3175%) = 2.5174.

6.3.3 Clock Rate

A well designed ASIC will always achieve a higher clock rate than a CCM device. This is a direct consequence of the known execution characteristics of the operations that are performed on the ASIC. A VLSI designer can "tweak" a design to lower capacitance, enlarge drivers and shorten wires to reduce propagation delays in ways that a CCM programmer cannot. The resulting higher clock rate in a microprocessor and custom ASICs gives them an advantage over the corresponding CCM implementation. The flexibility of the CCM gives it an advantage

over an ASIC, and the other factors in this section give the CCM advantages over microprocessors.

In the FIR example, the clock rate of the microprocessor was assumed to be 200 MHz as compared to Colt's mere 50 MHz. Thus, there should be a factor of 0.25 factor of Colt vs. the microprocessor. While it can be argued that 200 MHz is not the fastest microprocessor available today, it can also be argued that Stallion should run at 100 MHz.

6.3.4 Degrees of Concurrency

The RISC microprocessor assumed here contains only one execution unit, so that the degree of concurrency that is exploited is limited to 1. Colt, on the other hand, exploits a great deal of concurrency in evaluating the adder tree. The first 8 configurations perform 7 additions simultaneously, and the remaining two configurations perform 3 and 4 additions simultaneously. Thus, on average, Colt exploits $(8 * 7 + 3 + 4) / 10 = 6.3$ degrees of concurrency, which directly factors as an advantage over the microprocessor implementation.

6.3.5 Colt vs. Microprocessors Summary
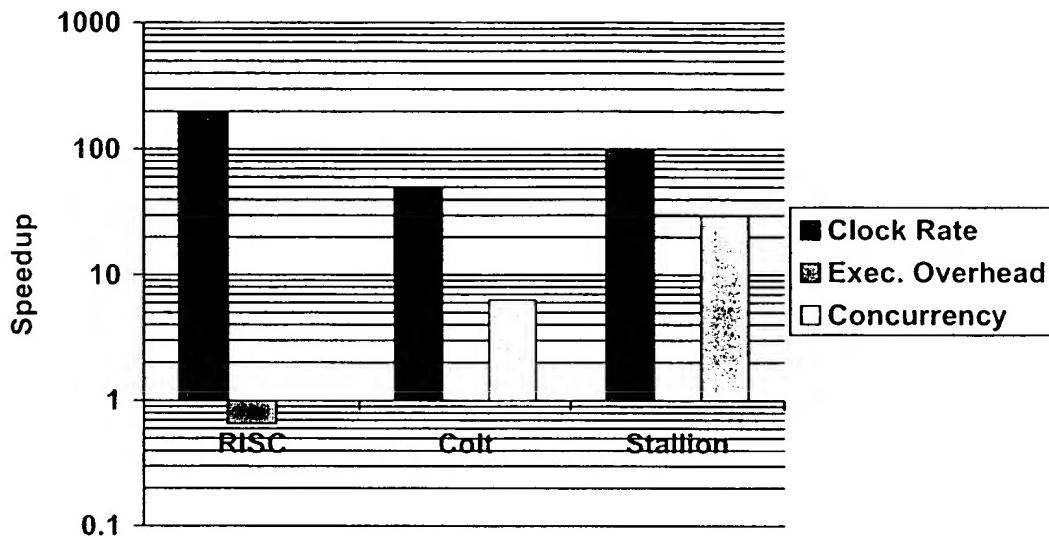
Figure 6.11- Colt/Stallion vs. Microprocessor Speedup Factor Comparison.

Table 6.3 shows a speedup factor of 3.713 of Colt over the microprocessor device. We have seen that Colt has a factor of 2.5174 advantage in terms of execution overhead, a factor of 0.25 in terms of clock rate and a factor of 6.3 in terms of degrees of concurrency. Together, these give an overall calculated speedup of 3.96, which is within 7% of the speedup actually attained. This would seem to indicate that these three factors are the dominant influences on the speedup attained by the Colt device.

As confirmation of the speedup equations presented in Section 4.1, they can be used to predict the speedup value that should result from this application. The equations are repeated here:

$$T_{CPU} = N_{ITER} \bullet \frac{T_{SETUP\_CPU} + T_{EXEC\_CPU}}{N_{UNITS\_CPU}} \qquad (4.3)$$

$$T_{CCM} = T_{SETUP\_CCM} + \frac{N_{ITER} \bullet T_{EXEC\_CCM}}{N_{UNITS\_CCM}} \qquad (4.4)$$

To calculated the predicted speedup, the window size of 10 ms will be fixed as the value for both

T_{CPU} and T_{CCM}, and the N_{ITER} parameter will be backed out of the equations. There are 59

addition operations to be performed; thus, T_{SETUP_CPU} is $59 * 1.52 = 89.68$ clock cycles.

Likewise, T_{EXEC_CPU} is 59 clock cycles, assuming one clock cycle per addition execution. This

results in $(89.68 + 59) * 5$ ns $= 743.4$ ns for the numerator term of the T_{CPU} exquation. The

N_{UNITS_CPU} is 1, giving a value for N_{ITER_CPU} of 13,451.

To predict N_{ITER_CCM}, the average setup value of 8.47 clock cycles is used for T_{SETUP_CCM}

times 59 adder configurations for a total of $59 * 8.47 = 499.73$ clock cycles spent in

configuration. Using a 20 ns clock, this results in $T_{SETUP\_CCM} = 20$ ns $* 499.73 = 9.995$ µs.

N_{UNITS_CCM} is dependent on the application and the implementation, and so must be taken from

the example to be an average concurrency of 6.3 operations per clock cycle. The value for

T_{EXEC_CCM} can be taken as 59 additions times one clock cycle per addition for a total of 1.18 µs.

These numbers give a predicted performance of $N_{ITER\_CCM} = 53,336$, for a predicted speedup over

the microprocessor of 3.96. This would seem to validate the style of analysis performed in

Chapter 4 and validates the speedup equations introduced there.

Table 6.6 - Comparison of Speedup vs. Microprocessor Results for FIR filter application.

Device	Speedup vs. 200 MHz RISC µproc		
	Actual	Calculated	Predicted
Colt	3.713	3.96	3.96
% Error	N/A	6.65%	6.65%

6.4 Colt vs. CCMs

A cursory glance at the area normalized results in Table 6.4 shows an increase in processing rate using Colt, and an even greater increase using Stallion. As with the microprocessor discussion, however, the area normalized results are difficult to quantitatively justify because of the wide variations in layout experience, design and process specifications between devices. The speedups over the competitors that have been achieved, even when layout area is taken into account, remains a testimony to a small, but determined, design team, the Colt/Stallion design and the Wormhole RTR concept.

Many observations can be made from the table. For example, the microprocessor approach achieves better performance per unit area than do the XC4000 series FPGAs. In fact, the XC4013E isn't even capable of meeting the real-time constraints on the problem due to a long reconfiguration time. The Xilinx XC6216, though a close competitor for Colt based on pure speed, falls behind when chip area is taken into account. The overall performance gains of Colt over the other CCM devices again are the result of a combination of many factors. A discussion of these follows. The XC4013E has been chosen as a baseline for purposes of comparison because of its similarity to the majority of FPGA devices on the market. All speedups shown will be relative to the performance of that device.

6.4.1 Configuration Time

As established in Chapter 4, faster reconfiguration times can result in better overall performance of an application running on a CCM. In that section, a comparison was drawn between reconfiguration time and the overhead of a microprocessor associated with cache misses, data dependencies, missed branch predictions, etc. In a similar sense then, reduced reconfiguration time can translate into higher performance. The key difference between the overhead of a microprocessor and that of a CCM is that the CCM overhead is incurred once per set of executions, whereas the overhead of the microprocessor will be incurred for every execution. Thus, when enough operands can be executed using a given configuration, the configuration overhead tends to become irrelevant. The number of operands required to reduce the overhead of the CCM device below that of a microprocessor, or even to the point of insignificance is of course dependent on the speed of configuration of the device.

There are many situations in which the number of operands are not large enough to entirely overshadow the effects of reconfiguration overhead. Real-time applications, as are commonly found in signal processing applications, often impose constraints on the maximum allowable latency for a given calculation. The FIR filter described in this chapter is an example of one of these. Another situation in which the number of operands may not be sufficiently large to dominate the configuration time would be in a virtual hardware situation when a small CCM is emulating a much larger hardware platform. Many configurations may be required in this situation, requiring frequent hardware page swaps. It may be that the number of data operands that can be processed using a given configuration is limited in such a way that it cannot be easily

extended through partial reconfiguration or other means. Vector data may exhibit this characteristic. Whatever the reason, configuration time can contribute to the speed of execution of an application and the improvements offered by the Colt device, and others, will be compared in this section.

6.4.1.1 Configuration I/O Rate

Table 6.7 - Peak Configuration I/O Rate Comparison.

Device	Number Of Configuration Pins	Configuration Clock Frequency	Configuration I/O Rate (Million Bits /Sec)	Speedup Relative To XC4000
XC4000	1	10 MHz	10	1x
XC6216	32	25 MHz	800	80x
Colt	96	50 MHz	4,800	480x
Stallion	192	100 MHz	19,200	1,920x

One component of reconfiguration time is the speed with which configuration data can be moved through the pins of the device. Running at 50 MHz, the six 16-bit data ports of the Colt can be used to simultaneously transfer 4.8 billion bits of configuration information per second through the I/O pins. In comparison, the Xilinx XC4000 series is capable of 10 million bits per second and the Xilinx XC6216 can transfer 800 million bits per second. Stallion is expected to have twelve 16-bit data ports, with a clock rate of 100 MHz, giving a peak I/O rate of 19.2 billion configuration bits per second.

For the FIR filter example, however, only one of the six ports is used to inject configuration information into the chip for the first seven configurations, which must configure 15 units each. The eighth input stage uses one port to configure seven units. The last two

218

configurations can make use of four ports to configure three units and five ports to configure four units, respectively. This gives an average of $(1 * 7 * 15 + 1 * 7 + 4 * 3 + 5 * 4) / 119 = 1.21$ data ports through which configuration information is being transferred. This reduces the I/O transfer rate to 968 million bits of information per second. This number gives a speedup of 96.8x over the XC4000 series for the example. The Stallion implementation would suffer from the same data port bottleneck, but it would be aided by a doubled clock frequency, giving an I/O transfer rate of 1,936 million bits of configuration bits per second for a speedup of 193.6x over the XC4000 series.

This speedup can be attributed to three major improvements of the Colt architecture over previous CCM and FPGA type devices. The first of these is the truly word-wide configuration path made feasible by the Wormhole RTR concept of transmitting configuration data over the same signal paths used for operand data. While a word-wide configuration path can be used to configure other devices such as the Xilinx XC6216, Wormhole RTR eases the burden of support by allowing data and configuration information to use the same signal path. This technique can save silicon area over the alternative of implementing twin bus structures, one for configuration and one for operand data.

The second contributing factor to the speedup is derived from the high clock rate possible due to the localized nature of communications within the Colt CCM. A global control strategy, such as a random-access based approach, requires a network of signals consisting of row and column decoders, driving buffers, etc., to allow any part of the chip to be accessed at any given time from the point of centralized control. By limiting the access to a localized region of the device, shorter propagation times can be achieved and higher clock rates realized.

219

The third benefit of Wormhole RTR that contributes to the performance of the Colt CCM is distributed control which allows all six data ports to be used to configure the device simultaneously. By alleviating the bottleneck of a single injection point for configuration data, all the advantages of parallelism are gained.

6.4.1.2 Configuration Storage Density

Table 6.8 - Basic Configuration Storage Density Comparison.

Chip	Configuration Bits Per Cell	Bits Output Per Cell	Config. Bits Per Output	Improvement Relative To XC4013E
Xilinx XC4000	430	2	215	1x
Xilinx XC6200	24	1	24	8.958x
Colt	128	16	8	26.87x

The configuration storage advantage can be quantified as shown in Table 6.8. Each Colt IFU requires a total of 116 configuration bits. The Xilinx XC6200 series of chips require 3 bytes of programming information per programmable cell. A cell is essentially capable of processing two single bit inputs and producing a single bit output. A third input can be used to multiplex the other two input signals, but the same thing can be performed in the Colt FU by the conditional unit, so this capability is equivalent. This results in 24 bits of programming information per processed bit in the XC6200 architecture. In the Colt architecture, there are 116 bits of programming information per 16 bits of processed output, giving 116/16 = 7.25 bits of programming information per bit of processed output. In terms of chip area consumed by programming overhead and in terms of programming time then, the Colt chip can be 24/7.25 =

3.31 times more efficient than the Xilinx part. A similar argument can be made for the Xilinx XC4000 series components, except that the advantages are even more dramatic.

The question arises of why Colt's configuration bit storage efficiency is limited to be only a factor of 3.31x greater than the XC6200 series. Intuitively, the Colt storage efficiency should be 16 times greater than the XC6200 because each Colt IFU contains the equivalent of 16 XC6200 cells all sharing the same configuration information. However, the Colt cells offer much more functionality than the Xilinx cells. For one, the Colt cells are designed to support the Wormhole RTR programming paradigm. Further, the ALU, *Conditional Unit*, *Barrel Shifter* and *Skip Bus* in every FU require more resources to control than the few that are in a XC6200 cell. The exact benefit of these, again, is hard to quantify, but they do contribute to the less than 16x improvement. Nonetheless, the application of these units to floating point operation, multiplication and conditional execution is undeniable.

The complexity of the operations of the ALU is apparent even in the FIR filter application. A single adder bit slice on the XC6200 requires four cells to implement because of the simplicity of the cells. This effectively divides the configuration storage density of all adder output bits by a factor of four on the XC6200. The loss of configuration storage density does depend on the application because, not surprisingly, the number of cells required to implement a particular function will depend on the function. The basic "cells" of all the other devices are sophisticated enough to implement an adder using a single "cell." In order to calculate a speedup penalty for the XC6200, the average number of cells per unit can be calculated as (4 * 59 adders + 1 * 60 buffers) / 119 units = 2.487 cells per unit. Thus, the XC6200 has an effective configuration storage density of 8.958 / 2.487 = 3.601x for the FIR filter application.

6.4.1.3 Broadcast Programming

The effect of broadcast programming on the total reconfiguration time is only a factor for the XC6216 in the FIR filter example. With the exception of Colt and Stallion, none of the other CCM devices support this concept. In the XC6216, broadcast programming mode can be used to simultaneously configure all the cells in a delay register or all the cells used to create an adder. At the device configuration level, the repetitive structure of the FIR filter example is limited to the bit level because the interconnections between word-wide units (delays and adders) must be customized to fit the situation. Because Colt and Stallion are word-oriented devices, and the FIR filter interconnections are not repetitive at the word level, their broadcast programming capabilities are not useful in this situation.

For purposes of comparison between devices, the effect that broadcast programming has on the configuration time of the XC6216 must be quantified. The extent to which broadcast programming can be used is an application specific parameter and so it will need to be factored into the calculated and predicted performance results as a fudge factor. The average number of physical cells programmed per write will be used. There are 60 12-bit registers, constituting 720 cells, which were all assumed to be configured using a single 24-bit configuration write cycle. It was also assumed that the full width of each of the adders could be configured using a single configuration write cycle. Without routing area, 3,282 cells are required to configure 59 adders. Each adder bit requires four cells to implement, but all like cells within the adder can be written simultaneously. Thus, $4 * 59 = 236$ broadcast writes must be done for the adder tree. This gives a total of $236 + 1 = 237$ broadcast writes to configure 4,003 cells. This reduces the effective number of cells that must be configured to 5.920% of the total, for a speed improvement of

16.89x over Colt and the XC4000 series. However, the broadcasts are using 24-bit writes, instead of the normal 32-bit variety. This will reduce the effective I/O transfer rate by a factor of 24/32, giving a speedup of 12.67x.

However, as was discussed in the FIR filter implementation example, the configuration bit addressing mechanism of the XC6216 requires additional bits to be written into the device to control the broadcasting and multicasting functions. The proportion of these that must be written relative to the number of cell configuration bits is, once again, application dependent. For that reason, the average number of clock cycles required per cell configuration will be taken from the example to be 2.25, based on that analysis. These control bits are embedded in the normal configuration bit streams of the XC4013E, Colt and Stallion and so are already included in the calculations for those devices. Relative to the XC4013E, Colt and Stallion, the XC6216 receives a 1 / 2.25 = 0.4444x speedup penalty, because of these additional control bits. This gives the final speedup due to broadcast programming of 12.67 * 0.4444 = 5.63x.

6.4.1.4 Partial Run-Time Reconfiguration

Partial run-time reconfiguration is supported by the XC6216 as well as the Colt and Stallion CCMs. The configuration granularity of the other devices is limited to the complete chip level and the configuration time for those is a simple matter of dividing the number of cells required for the application by the number available on the chip, rounding up and multiplying by the full chip reconfiguration time. Partial Run-Time Reconfiguration can be used to improve the reconfiguration time in two ways. First, only the computational elements that are actually used on the device need to be configured. The configuration time for a partially configurable device is

then reduced to the number of computational elements used multiplied by the configuration time of a single unit. Second, once a configuration has been programmed, the programmer of a partially configurable device has the option of modifying the existing configuration to match the next computation rather than discarding the existing configuration and starting anew. This ability was used in the XC6216, Colt and Stallion implementations of the FIR filter by reusing the structure of the adder tree below the first level. These two effects combine to improve the overall configuration time.

For the partially configurable devices, an approximation of the reconfiguration time saved can be made by first noting that there are 119 units in the FIR filter application, and then taking into account the number of these that must be actually configured. For the XC6216 and Stallion chips, 15 of the first level adders are implemented in each configuration. During the second configuration, these 15 can be partially reconfigured and the rest of the configuration can be reused. Thus, $119/2 + 15 = 74.5$ units must be configured in total, meaning that only $74.5/119 = 62.6\%$ of the actual number of units used must be configured onto the devices, giving a speedup of 1.597x. For Colt, seven of the eight configurations used to implement the input row can be partially configured. The first seven configurations used in the input row contain four first level adders out of 15 units each. The last input row configuration could be implemented by reconfiguring the first two first level adders and the third level adder. Hence, there are 15 units in the first input row configuration, $6*4 + 3 = 27$ units that must be configured in the rest of the input row, and 7 units in the two collection configurations. This results in $(15 + 27 + 7)/119 = 41.18\%$ of the units of the FIR filter structure must be configured for a speedup of 2.428x.

Table 6.9 - Configuration Time Speedup Due To Partial Reconfiguration.

Device	# Of Units	# Of Reusable Units	% Cells Requiring Configuration	Speedup Relative To XC4013E
Xilinx XC6216	119	44.5	62.60%	1.597x
Colt	119	70	41.18%	2.428x
Stallion	119	44.5	62.60%	1.597x

Note that the greatest contribution to speedup from partial reconfiguration was given to Colt. This is due to the fact that Colt's units are reused across more configurations than for the other implementations. This could be used as an argument to use smaller CCM devices in place of larger ones, or it could be used as an argument to configure smaller portions of a larger CCM device and use the saved resources for other tasks. A balance could be struck between the speedup gained through the reuse of a small number of resources in a given configuration versus the speedup gained through the greater parallelism, and possibly longer configuration times, of using a large number of resources.

6.4.1.5 Routing Overhead

The routing overhead associated with a programmable device is dependent on the application and the routing resources available in the architecture. In most cases, some computational cells must be sacrificed as routing resources in bit-oriented architectures such as the XC4013E and the XC6216. The exact percentage of the cells that must be sacrificed in this way varies; however, from lab experience and actual usage, we conservatively estimate that 25% of the cells required for a fully implemented design would be used as routing resources. Colt, being a word-oriented architecture with improved routing resources, does not require any

computational resources to be sacrificed in order to implement the FIR filter example. This gives an effective speedup of $1/(1 - 25\%) = 1.333x$ to Colt in terms of the number of cells that must be configured to perform the computation.

6.4.1.6 Bit-Sliced Design Advantage

Because the XC4013E and the XC6216 are both bit-oriented architectures, they can both construct adders and registers that are the exact width required at any given point in the computation. The delay stages can be made 12 bits wide, and the adders start at 13 bits and grow successively wider. All delay stages and adders in Colt and Stallion are 16 bits wide because that is the word width chosen for the architecture. This causes waste in that Colt and Stallion are configuring bits that are not required for the computation. There are 60 delay stages, each of which is 16 bits wide and only needs to be 12 bits wide, meaning that $4 * 60 = 240$ bits are configured unnecessarily. From Table 6.2, the number of unnecessary adder bits is $30 * 3 + 15 * 2 + 7 = 127$, for a total of 367 "bits" that must be configured, but are not needed. A total of $60 * 12 = 720$ are needed for the buffers and $30 * 13 + 15 * 14 + 7 * 15 + 7 * 16 = 817$ for the adders, giving $720 + 817 = 1,537$ bit operations that are required. Thus, Colt and Stallion must take a penalty of $1537 / (1537 + 367) = 0.8072x$ in terms of wasted resources.

6.4.2 Execution Speed

Obviously, the speed of execution of a given configuration plays a large part in determining the overall system throughput. The factors that affect the speed of execution will be

226

discussed in this section and speedups will be calculated relative to the XC4013E implementation of the FIR filter.

6.4.2.1 Clock Frequency

The operating frequency of the various designs directly affects the speedup that can be obtained. Since all of the FPGA designs implemented some form of pipelined adder tree, an increase in clock rate would result in a proportional increase in speed. The clock rate of the FPGA implementations generally suffers at the low rate of 10 MHz due to long propagation delays through the programmable routing resources available. Colt can gain an advantage over other CCM systems because of the coarse grained computation approach. The Xilinx XC4013E contains 576 CLBs on a die that is 944 million λ^2, meaning that 1.64 million λ^2 is devoted to each CLB. Colt contains 16 IFUs plus a multiplier on a die that is 382 Million λ^2, so that approximately 22.5 million λ^2 has been allocated for each computational element. Similar comparisons can be made with other devices. Because the allocation units on Colt are larger, there are fewer units to connect (17 compared to 576), meaning that fewer switch points need to be used. Fewer switch points translates to less propagation delay. Further, because there are fewer connection paths, and fewer buffering stages are required, the buffers that do exist can be made larger so that the delays along the paths are shorter. Both of these affects contribute to a higher operating frequency for the Colt CCM.

Table 6.10 - Operating Frequencies of FIR filter Implemenations vs. XC4013E.

Device	Execution Clock Rate	Speedup vs. XC4013E
XC4013E	10 MHz	1x
XC6216	10 MHz	1x
Colt	50 MHz	5x
Stallion	100 MHz	10x

It is impossible to directly compare the clock rate of the various FPGA implementations without detailed knowledge of the VLSI construction of each. Indeed, even with such knowledge, the estimation of operating speeds is often speculative at best. For these reasons, the manufacturer's estimated clock rates for the various implementations will be used to calculate speedup values. As was mentioned above, the Xilinx XC4013E and XC6216 implementations are both estimated to operate at a maximum clock frequency of 10 MHz. Based on our analog simulations of the Colt CCM, we estimate a lower bound of 50 MHz for its operation. Through extrapolation and further pipelining of the design, the clock rate of Stallion is estimated to be 100 MHz. These values are summarized and the speedups relative to the XC4013E are shown in Table 6.10.

6.4.2.2 CCM Die Size vs. Number Of Configurations

The size of a CCM directly affects how many configurations are required to implement a particular function. In the FIR example, ten reconfigurations were required for Colt, whereas most of the other CCM devices required only two reconfigurations. There are several important points that should be made about this tradeoff. First, for devices such as the Xilinx XC4000 series that cannot be partially configured, a large device can be detrimental. The long

228

configuration time of the XC4013E prevented it from performing even one evaluation of the FIR filter within the 10 ms window.

If the device can be large and still maintain a low reconfiguration time, such as is the case for the XC6216 and for Stallion, then an improvement in performance can be obtained. Fewer reconfigurations not only translate into less time spent in the reconfiguration process, but can also result in greater degrees of parallelism. Since more of the computation can be directly implemented in hardware, more of it can proceed concurrently. The clock frequency of the device must be divided by the number of configurations to obtain the effective clock rate of the "virtual device" that is being simulated by the CCM hardware. The fact that the normalized processing speed of Stallion is greater than that of Colt is a direct result of the larger die size and also of the higher clock rate that Stallion will employ.

A larger chip approach may rely on the fact that enough of the application can be parallelized to fill the available resources. If this is not the case, then the application is limited by Amdahl's Law. The XC6216 has a single control port through which it may be reconfigured. A single configuration point makes it difficult for multiple controllers to share the resources of a large device. Stallion, by virtue of Wormhole RTR, has multiple control ports. These can be used simultaneously, making it easier for several controllers to use the resources simultaneously. Thus, though a given application may be limited by Amdahl's Law, Stallion's ability to easily support the configuration of multiple applications simultaneously may make the deleterious effects of using a larger die more manageable by allowing these applications to share the resources.

Finally, in a large system where the application is much larger than the size of the device implementing it, many reconfigurations will be required. In these situations, the reconfiguration time of the device will become a percentage of the overhead, just as incorrect branch predictions, cache misses, etc., are for microprocessors today. It is then important to minimize the time spent in the reconfiguration process. Devices based on a scheme such as the XC4000 series, whose configuration time is a function of only the number of functional units on the die and is independent of the number of I/O pins available, will grow to have proportionally longer configuration times as die sizes increase.

Table 6.11 - Number of FIR Filter Configurations Speedup vs. XC4013E.

Device	# Of Config.	Speedup Relative To XC4013E
XC4013E	2	1x
XC6216	2	1x
Colt	10	0.2x
Stallion	2	1x

For purposes of the FIR filter example, each implementation suffers a speedup penalty that is equal to the inverse of the number of configurations required. The XC4013E requires two reconfigurations, as do the XC6216 and Stallion, giving all of these a speedup relative to the XC4013E of 1x. Colt, on the other hand, requires ten configurations, giving it a penalty of 2/10 = 0.2x relative to the XC4013E.

6.4.3 Colt vs. CCMs Summary
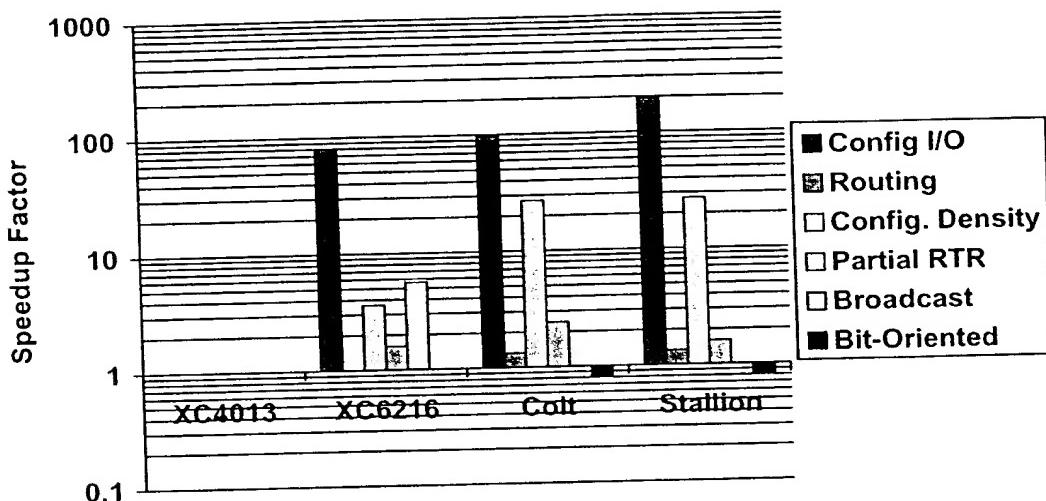
Figure 6.12 - FIR Configuration Speedup Factor Comparison.

Table 6.12 - Calculated Configuration Time Speedup Factors vs. XC4013E.

Configuration Speedup Effect	Relative Speedup By Device			
	XC4013E	XC6216	Colt	Stallion
Config. I/O Rate	1	80	96.8	193.6
Routing Overhead	1	1	1.333	1.333
Config. Storage Density	1	3.601	26.87	26.87
Partial Run-Time Reconfiguration	1	1.597	2.428	1.597
Broadcast Programming	1	5.63	1	1
Bit-Oriented Design	1	1	0.8072	0.8072
Speedup	1	2,590	6,795	8,939
Calculated Config. Time	N/A	19.15 μs	7.30 μs	5.55 μs
Actual Config. Time	49.6 ms	19.8 μs	10.42 μs	6.42 μs
% Error	N/A	3.3%	30.0%	13.5%

231

All the effects contributing to configuration time that have been discussed are summarized in Table 6.12. The product of the various factors has been listed as an overall speedup vs. the XC4013E implementation. Dividing the XC4013E's configuration time of 49.6 ms by the speedup for a device gives an approximation for the configuration time of that devices. This approximation has been listed as the calculated configuration time. This has been compared against the actual configuration time that was derived and the percentage error in the calculation is shown. It is important to note that there are at least six different parameters that contribute to the overall speedup and that if each of them were accurate to within 10%, then the resulting speedup should be accurate to within $1 - (1 - 0.1)^6 = 46.8\%$. This accounts for the rather large variance of the calculated value for the Colt device.

Given that the speedup values appear to be an accurate approximation of the combined effects of the various factors on the configuration time of the devices, observations can be made about the relative importance of these. The configuration I/O rate would appear to be the single largest contributor to speedup. All three of the devices being compared have significantly more configuration I/O bandwidth than the XC4013E. Interestingly, the I/O rate achieved for this application by Colt and Stallion is near the low end of their possible range. This is because, on average, barely more than one data port is used to configure the devices for this application. More I/O ports could have been used to more quickly configure the adder tree structure on the devices and these would have significantly affected the overall configuration times, as should be evident from the peak rates listed in Section 6.4.1.1.

Another significant boost to the speed of Colt and Stallion comes from the configuration storage density factors. These numbers are a direct result of the word-oriented approach that was

232

taken in the design of these devices. The word-oriented approach allowed the same configuration bits to control the functions performed on 16 processed outputs. Further, the larger computational units allowed more functionality to be placed in each unit; thus avoiding the problems of simplified cells found in the XC6216 where four cells are required to construct an adder bit slice. The draw back of the word-oriented approach that has been described is that resources may be wasted due to the relatively course grain allocation that is required. The bit-oriented design factor indicates that less than 19% of the resources have been wasted in this way. Naturally, this is an application specific parameter, but Colt and Stallion are targeted toward DSP type applications where word-oriented processing is more the norm.

Partial run-time reconfiguration played a smaller part in the overall speedup achieved by all the devices, but this could be due to the size of the devices being employed. As VLSI technology advances, the silicon available to designers will continue to increase. The ways in which this increased area should be used is a subject of debate. As for CCM devices, it seems clear that a configuration strategy that allows partial configuration of a larger device would be superior to an all-or-nothing approach. For this application, the speedup gained varied from 1.6 to 2.4, which, though significant, could be dwarfed by the contribution of this capability in larger devices.

Broadcast programming contributed a significant part of the speedup of the XC6216 device. A factor of 5.63 was gained over all the other CCMs in the comparison. This can in part be attributed to the simplicity of the cells used in the XC6216, because many more identical cells (a factor of 2.5 in this case) were required on average to implement the application. Thus, these lend themselves to a broadcast programming approach at the bit level. Colt and Stallion, on the

other hand, consist of fewer, more complex computational elements that, in effect, already

benefit by such broadcast programming because of the conglomeration of 16 computational

outputs into a single unit being controlled by the same configuration information. As was noted

earlier, this more coarse grained approach also gains advantages in terms of the routing resources

that can be implemented, which then reduce the number of computational resources that must be

sacrificed to routing. In a sense then, the XC6216 can leverage a large factor for broadcast

programming at the expense of the routing overhead and configuration storage density factors.
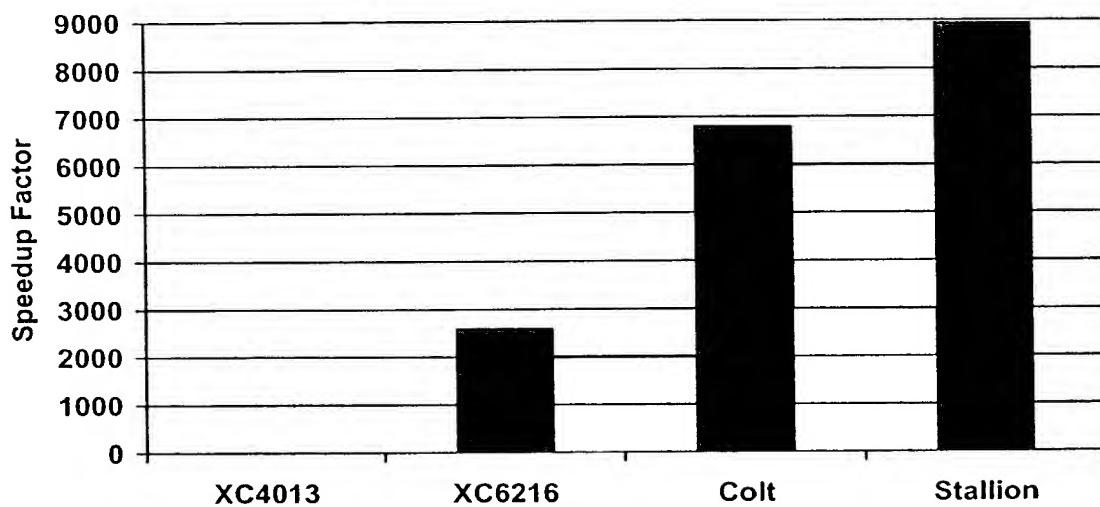
Figure 6.13 - Overall FIR Configuration Speedup Comparison.

Finally, the configuration speedup time of Stallion over Colt should be discussed. In part,

the small improvement is due to the same data port bottleneck that hampered the configuration

I/O rate in relation to the XC6216, and again here, the doubled number of data ports in Stallion

could be used to improve this performance. The other significant difference between the two

234

chips is derived from the partial run-time reconfiguration speedup that Colt gains. The fact that proportionally more of Colt's resources can be reused without reconfiguration is an advantage. As discussed in Section 6.4.1.4, the tradeoff of configuration reuse vs. parallel implementation should be a significant concern for CCM technology compiler developers.

Table 6.13 - CCM Reconfiguration Time Ranges.

Chip	Effective Pins Useable For Reconfiguration Per Clock Pulse	No. Bits Required	Reconfiguration Time (Complete)
Altera FLEX 8000	N/A	N/A	100 ms
Xilinx XC4013E	1 (10 MHz)	247,960	24.8 ms
Xilinx XC4025	1 (10 MHz)	422,168	42.2 ms
Xilinx XC6216	32 (25 MHz)	160-98,304	200 ns to 122.88μs
Colt	96 (50 MHz)	64-2,176	140 ns to 2.72 μs
Stallion	192 (100 MHz)	64-8,320	70 ns to 5.20 μs

It is useful to examine the range of possible configuration times for these devices as a way of comparing worst cast times that could arise for various applications. Determining this parameter is simple for earlier FPGA devices such as the Altera FLEX 8000 series or the Xilinx XC4000 series in which the entire chip must be reconfigured at once. The possibility of partial reconfiguration does not exist. However, other devices, such as the Xilinx XC6200 series and the Colt do allow partial reconfiguration, and other features such as broadcasting that result in variable programming times. Table 6.13 shows the relevant parameters for a representative set of devices.

The wide spread of configuration times for the XC6216 is due to its broadcast programming mode. As discussed in Section 2.1.3.4, the chip can broadcast the same

235

configuration data to a large number of cells simultaneously. This would be useful for programming regular structures such as multipliers. In the extreme, the same configuration can be sent to all cells on the chip, giving the low end of the range. At the other end of the extreme, it is assumed that every cell in the chip requires a different configuration, or that the cells are placed on the chip in such a way that broadcasting cannot be used to programming like cells simultaneously. This situation could arise when configuring control circuitry. In reality, neither extreme is likely to occur often and the practical time for most applications will fall somewhere in between.

The wide range in configuration time for the Colt chip is also related to usage. At the low end of the range, the Colt is simply used as a multiplier. The longest path in that case would be a stream coming into a port, through the crossbar, through the multiplier, back to the crossbar and out a data port. This path would take 7 clock cycles to complete from pin to pin. The same path could be used for Stallion. The high end of the range is exemplified by a single stream entering the chip and programming the entire mesh and multiplier. This stream would program 16 IFUs, 2 data ports and six crossbar connections. This path would require 16 * 8 + 2 + 6 = 136 clock cycles, requiring 136 clock cycles / 50MHz = 2.72 μs. In the case of Stallion, this path would require programming 64 IFUs, giving 64 * 8 + 2 + 6 = 520 clock cycles, for a time of 520 / 100 MHz = 5.20 μs.
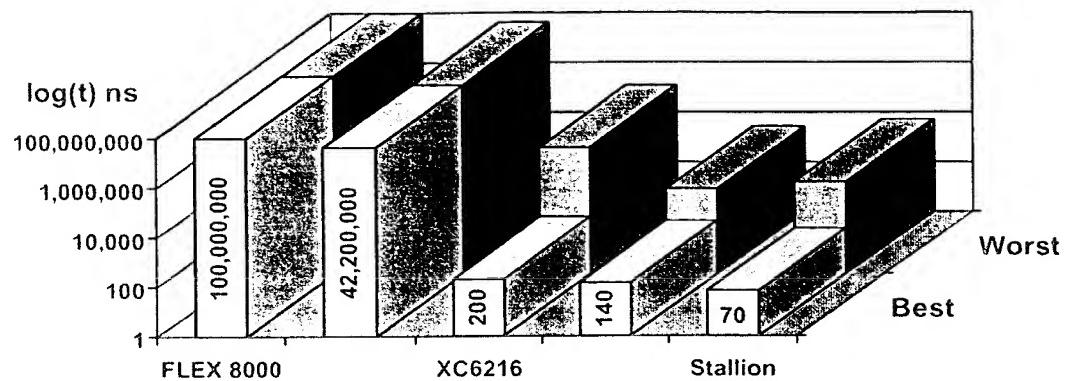
Figure 6.14 - Reconfiguration Time Comparison.

Table 6.14 - Colt Reconfiguration Times Relative To Other FPGAs.

Chip	Reconfiguration Time (Complete)	Colt Relative Speedup
Altera FLEX 8000	100 ms	36,765x to 714,286x
Xilinx XC4013E	24.8 ms	9,118x to 177,142x
Xilinx XC4025	42.2 ms	15,515x to 301,428x
Xilinx XC6216	200 ns to 122.88 μs	1.43x to 45x
Colt	140 ns to 2.72 μs	1x
Stallion	70 ns to 5.20 μs	0.5x to 1.91x

Figure 6.14 and Table 6.14 show the relative speedup of the various devices vs. the Colt reconfiguration times. Most of the speedups are quite dramatic, which can be partially attributed to the high configuration data I/O rate discussed in Section 6.4.1. An even greater improvement is shown in the overall configuration time because fewer configuration bits pass across the pins. This is due to the word-wide configuration strategy used by the Colt in which the operation

237

performed on an entire operand word is dictated by the same set of configuration bits. The other devices listed in the table are bit-oriented, requiring that configuration data be downloaded for each individual bit, even when they are identical, as would be the case for most DSP operations. Thus, the I/O transfer rate of the Colt is magnified by the lower configuration overhead and broadcast programming effects to produce an even greater effect on the overall configuration time. A comparison of Stallion vs. Colt shows that the doubled clock rate on Stallion can allow it to be configured in half the time for the low end of the range. At the high end of the range, Stallion must configure four times as many IFUs, but the doubled clock rate holds the configuration time to only twice that of Colt.
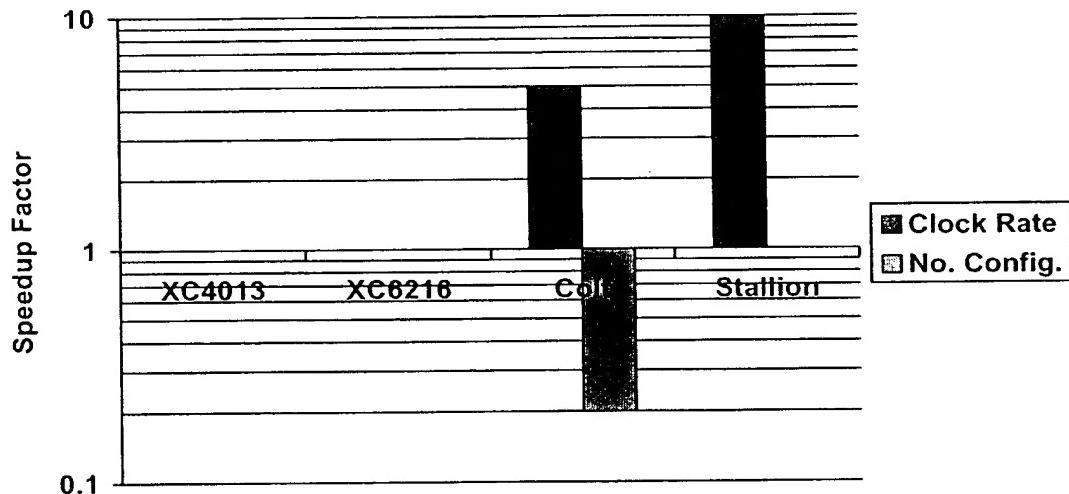
Figure 6.15 – FIR Execution Speedup Factor Comparison.

238

Table 6.15 - Calculated Execution Speedup Factors vs. XC4013E.

Execution Speedup Effect	Relative Speedup By Device			
	XC4013E	XC6216	Colt	Stallion
Clock Rate	1	1	5	10
# Of Configurations	1	1	0.2	1
Speedup	1	1	1	10
Calculated Execution Time Per Evaluation	200 ns	200 ns	200 ns	20 ns
Actual Execution Time Per Evaluation	200 ns	200 ns	200 ns	20 ns
% Error	0%	0%	0%	0%

The results of the comparison of the time spent in execution are summarized in Table 6.15. The dominant contribution to the overall performance of the devices on the application is the execution rate. The 10 ms window allowed enough time for all but the XC4013E to perform tens of thousands of evaluations of the FIR filter. This fact caused the differences in configuration time to be overshadowed by the shear quantity of evaluations that were possible. If fewer evaluations were possible, the effect of configuration setup time would have been more keenly felt. The execution clock rate that is made possible by the word-oriented approach of Colt and Stallion allowed them to sustain the greatest throughput of all the devices. While the XC6216 appears to be a close competitor to Colt in Table 6.3, the area normalized results of Table 6.4 show that this is due to the XC6216's use of nearly three times as much silicon to perform the calculations. When compared to the performance of the more similarly sized Stallion, the XC6216 suffers.
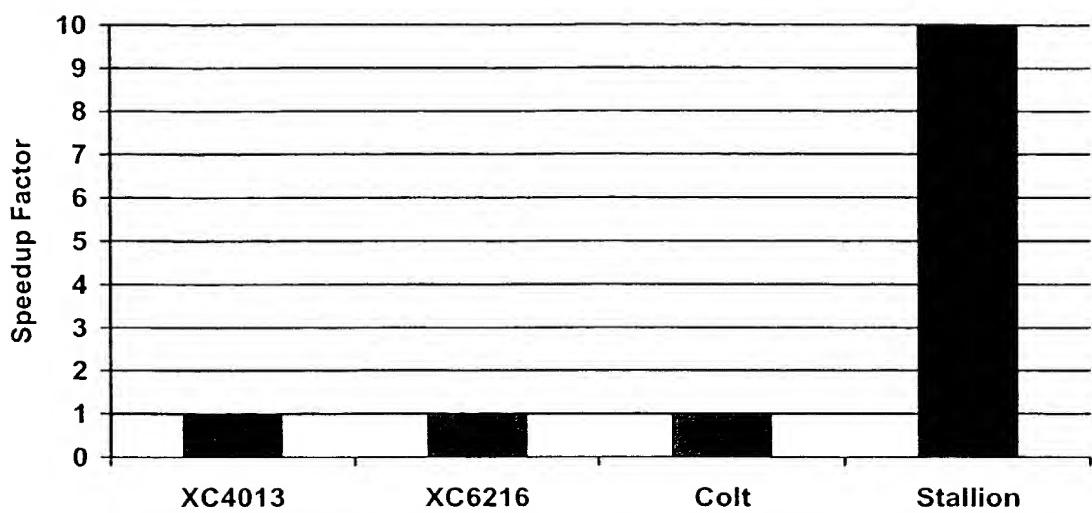
Figure 6.16 - Overall FIR Execution Rate Comparison.

7. Conclusions

As with any large design project, the creation of Colt has spurred many ideas for future work. Most of this chapter consists of descriptions of these possible improvements and areas of investigation for continued research. The chapter concludes with a brief discussion of possible applications to which the Colt is well suited, along with some final words by the author.

7.1 Future Enhancements

The Colt is a prototype exemplifying the core concepts of Wormhole RTR. It is not intended as the sole method of implementation of the Wormhole RTR paradigm. There are many possible variations on the theme that would be better suited to different processing tasks. Even as a DSP processor the Colt could benefit from several additions and changes. It was built as a prototype to work out the conceptual and practical problems with such a design so that a follow-on chip, called Stallion, could be built that will incorporate the basic concepts of Colt while making enhancements in the design. Stallion is intended to be a much larger processor that exemplifies a greater wisdom. Though at this time Colt has only recently been submitted for fabrication and real silicon has not yet returned, the lessons of the practical design, implementation and early algorithm mapping have been many and it is the goal of this section to illustrate these.

241

7.1.1 Flip Flop Design

The present flip flop design, while small, does suffer from current leakage due to the use of NFET pass transistors to perform the latching function. The collective current drain from these is a significant source of power consumed by the chip. A different design should be considered for the Stallion processor. One possibility are the flip flops used in the DEC Alpha processor. These are 10 transistor devices, but they only require a single clock signal and they have negligible quiescent power dissipation.

7.1.2 Super-Pipelining

Currently, the critical path for execution is through the FU; in particular, along the carry path of the ALU. The simulated worst case delays through the ALU are on the order of 7 ns. The network propagation delays from the output of the ALU to other points in the circuit is of the same order. Thus, the clock frequency could easily be doubled simply by placing pipeline registers at the ALU outputs. Naturally, this would require pipelining of the other units, in particular the crossbar and multiplier; however, this is also easily achievable. The only drawback to this decision would be that in order to take advantage of the higher clock rate, the words of each stream would need to be presented to the pins at the rate of 100 MHz as well. This may not be a practical design strategy at the board level and should be carefully considered before pipelining the chip any further.

7.1.3 Relative Addressing

It may be advantageous to consider a relative addressing scheme for units on the chip. This would shorten the number of bits need for the addressing units on the chip. Instead of giving each unit a chip-wide unique address, unit locality could be used to specify cardinal directions in the mesh or one of the outputs in the crossbar, etc. The disadvantages, as previously discussed, would relate to divergent or split streams and also to broadcast programming. Both of these problems could be solved by the use of special "split" packets that would effectively act as markers within the stream header to indicate separate sections of the header, each of which could proceed down a different direction of the split or broadcast divergent point. Such a design may be more efficient in terms of the total number of configuration bits required in some situations.

7.1.4 Unit Stream Header Response

As discussed in Section 4.9, all units should be made to function as the FUs do with regard to accepting programming information. The stream header should not be allowed to pass through a given unit until that unit has been explicitly programmed. This would aid in controlling divergent streams when the situation requires this type of use. Such a change should be a natural outgrowth of a relative addressing scheme.

7.1.5 Pipeline Stalling

A new system should be put in place that would allow computational pipelines to be stalled. This would allow a much greater degree of flow control than is possible using the

present scheme. For example, presently it is not practical to attempt flow control when directly connecting multiple Colt chips because of pipeline flushing when a "wait" signal is received. Adding a stalling mechanism to the on-chip pipeline would eliminate this shortcoming.

The stalling mechanism could be implemented by including a global set of control lines that go to every unit on the chip, much as the set that runs between the data ports in Colt. There would only be a need for a few of these. A unit in a pipeline would be programmed to watch the value on one of these lines. If that line is deasserted, it would stall. The control lines could be dynamically allocated just as the other hardware resources are. These lines would need a great deal of global routing and would need to have a distribution tree only slightly smaller than that used for the clock. However, as a practical matter, the number of such lines needed would be equal to the maximum number of pages that would simultaneously exist on the chip.

7.1.6 Stream Packet Format

The conventions used for the stream packet format on the Colt dictate that bit 15 of every word in the stream header is used as a flag to indicate whether or not it is the first word in the packet. In hindsight, this seems to have been a waste of the most precious commodity in the design: I/O resources in the form of the number of configuration bits that must be moved across the pins of the chip. It would be far better to use bit 15 as a normal configuration data bit and instead put some type of counter in each unit to monitor the length of each packet. This scheme fits much better with the local complexity over global routing theme.

The only complication to this scheme occurs when a unit that has not yet been programmed receives a packet for a different unit type. This could happen in the current scheme

when a split in the stream is used to program multiple paths using the same stream source because in Colt a split will forward the entire stream header in both directions. However, if the addressing scheme discussed in Section 7.1.16 were used, the stream header section markers could be used to direct the various sections of the header only down the paths for which they were intended; guaranteeing that this situation never occurs.

7.1.7 Crossbar Multicasting

In the Colt, the crossbar programmer has three options: use the existing configuration, program the crossbar to use a single output, or broadcast to all eight inputs to the top of the mesh. It is possible to broadcast to all eight mesh inputs and then selectively reprogram some outputs to receive data from another source at a later time; however, the timing of this option is tricky. Instead, the extra configuration bits that are available in the crossbar configuration packet should be used to implement multicasting. This could be done using a variety of methods, one of which would be to use the extra bits as markers that would program crossbar switches to accept succeeding programming packets as configuration information after the arrival of the first stream header packet. Care would need to be taken when implementing this addition to ensure that it doesn't have negative consequences in cases where multiple Colt chips were directly connected. The problems associated with addressing in this scenario were discussed in Section 4.9.

7.1.8 Crossbar Output Circuitry

The output columns of the crossbar in the Colt are designed using Pseudo-NMOS circuitry in order to avoid a short circuit for the case where two crossbar inputs simultaneously

program the same column as an output. Although a great deal of design effort was taken to reduce the current drain of these lines, the amount of power dissipated by these circuits in Colt is unacceptable. The Pseudo-NMOS circuitry should be eliminated and a different means of preventing this condition should be devised for all future generation chips. One possible method would be to prioritize the inputs to the crossbar in much the same way that programming directions are prioritized for the IFU, as discussed in Section A.6.2. The ClearOC line could be chained from one input row to the next so that it would only allow a given row to take control of the output column on the following clock period if none of the rows above it were attempting to do so. Of course, propagation delay through the chain could become a factor for large numbers of crossbar inputs and this would have to be carefully designed.

7.1.9 Mesh Sizing

The mesh on the Colt is four IFUs on a side. This size was chosen because of the layout dimensions of an IFU and the small die size that was originally targeted for the prototype. A smaller die size was chosen to keep down manufacturing costs for the first run of chips, allowing us to get back chips for testing that exhibit all the functionality, albeit on a smaller scale than would eventually be desirable. The crossbar and mesh of the Colt as implemented require approximately $(4 * 8) * 156,750 + 16 * 11,440,000 = 188,056,000 \ \lambda^2$. With 16 IFUs on the chip it would have been more area efficient to create a chip using full crossbar connectivity between the IFUs; totally abandoning the mesh concept. This would have required a crossbar of 32 outputs and 16 inputs. The resulting crossbar would be 16 times the size of the 4 input, 8 output crossbar that was implemented. However, the FU constitutes only 18.96% of the total area

246

consumed by the IFU, so area would be saved by removing the mesh connectivity. For a Colt sized chip, a full crossbar implementation would require approximately (16 * 32) * 156,750 + 16 * 2,169,750 = 114,972,000 λ^2. Thus, the area requirement of the full crossbar implementation is only 61.14% the size of the actual Colt design. The advantages of the increase in connectivity can only truly be appreciated through actual experience in mapping a complex algorithm to the mesh. It should be noted that these computations neglect the routing from the crossbar to the FUs. It is hoped that this routing could be run in metal 3 over the FUs so that the impact would be minimal.

For a larger chip, such as Stallion, the full crossbar scheme becomes unfeasible due to the large numbers of FUs that would be present. Assuming that Stallion contains 64 FUs, the crossbar would need to have 64 inputs and 128 outputs, minimum. The area requirement for a full crossbar on the Stallion is approximately (64 * 128) * 156,750 + 64 * 2,169,750 = 1,422,960,000 λ^2. An 8 by 8 mesh of IFUs, connected to the crossbar in a manner similar to Colt would require (8 * 16) * 156,750 + 64 * 11,440,000 = 752,224,000 λ^2. Thus, a Stallion architecture incorporating the extended mesh could be implemented in 52.86% of the area required for a full crossbar implementation. Though the full crossbar is convenient, the n^2 nature of its growth becomes prohibitive for large sizes. Because the Colt is to some extent mirror the architecture of Stallion, the mesh was included.

In the actual Stallion implementation, a balance could be struck between the full crossbar and partial mesh implementations based on the area requirements of the die. There are a number of considerations to be taken into account when making this tradeoff. For one, the number of

247

Skip Bus segments that must be traversed from the top of the mesh to the bottom should be considered. While the amount of time needed for a signal to propagate through a Skip Bus segment is not long, the time is cumulative, and, for large numbers of skips, can be significant. There are cases, such as the factorial calculation example, in which it is advantageous to propagate a signal from the top of the mesh to the bottom row of FUs using the Skip Bus. For that reason it may be wise to avoid overly deep mesh configurations, using a 4 high by 16 wide IFU mesh instead of an 8 by 8 arrangement, for example. This would dictate a crossbar of 16 inputs and 32 outputs, thus the total area consumed would be $(16 * 32) * 156,750 + 64 * 11,440,000 = 812,416,000 \lambda^2$, which is only 8% larger than the 8x8 mesh implementation.

Perhaps the best possibility is to replace the crossbar as currently designed with a multistage network that retains the full crossbar connectivity while using a longer communication latency so that it can be implemented in half the area of the current crossbar. Due to the pipelined nature of the Wormhole RTR concept, the extra latency in the overall computation should not significantly affect performance, but a great savings in silicon area could be realized. A Stallion sized chip using this new network would require approximately $(64 * 128) * 0.5 * 156,750 + 64 * 2,169,750 = 780,912,000 \lambda^2$. This is only 3.8% larger than the 8x8 mesh implementation, while it offers vastly enhanced connectivity options.

248

Table 7.1 – Communication Network Sizing Strategies.

Chip	Crossbar Dimensions	Mesh Dimensions	Area (λ^2)
Colt (16 FUs)	4x8	4x4	188,056,000
Colt (16 FUs)	16x32	None	114,972,000
Stallion (64 FUs)	8x16	8x8	752,224,000
Stallion (64 FUs)	16x32	4x16	812,416,000
Stallion (64 FUs)	64x128	None	1,422,960,000
Stallion (64 FUs)	64x128 (Multistage)	None	780,912,000

The various implementation strategies and corresponding area requirements are summarized in Table 7.1. The most promising strategy appears to be the multistage network implementation shown on the last line, and its design should be carefully considered as a replacement topology to be used on Stallion. In hindsight, it would have been beneficial to have this data for the design of Colt, but real-world numbers such as these can only be attained by going through the full design cycle. The layout team, Mark Musgrove (IFU) and Dr. Peter Athanas (Crossbar), has provided the data to allow a proper analysis to attain an optimal communications network strategy. The full design of the Colt device as a prototype has proven its worth as a stepping stone to greater design wisdom.

7.1.10 Skip Bus Routing

A suggestion related to crossbar/mesh sizing would be to allow two outputs through the bottom of the mesh instead of just one. This would cause the size of the crossbar to double, but as can be seen from the calculations, it would not significantly impact the size of the die. There seem to be a reasonable number of circumstances under which it is desirable to propagate more

than one output out the bottom of a column. There are several options available to implement this ability, but one method would be to treat the Skip Bus nodes as independent units separate from the IFUs. Currently, all configuration information for the Skip Bus connectivity is stored in the IFU information. Configuration information cannot be transmitted through the Skip Bus and routing for a given segment is established by configuring an adjacent IFU. Instead, the Skip Bus could be considered as an independent path capable of routing and being configured by its own configuration data from the stream header. When a stream header leaves an IFU it would then have the option of proceeding along eight different paths: the four local directions currently supported, as well as four new paths offered by the surrounding Skip Bus segments. If the stream header proceeded along a Skip Bus segment it would arrive at an adjacent Skip Bus node and could then either configure the corresponding IFU, or proceed immediately along another Skip Bus segment without interfering with the configuration of the IFU in any way.

This alternative would seem to be a superior scheme to that implemented by Colt since it would allow the Skip Bus outputs from the bottom of the mesh to act as fully functional outputs. It would also allow the Skip Bus inputs at the top of the mesh to act as fully functional inputs; carrying configuration information as easily as the local connections. As can be seen from the factorial example, this ability would be valuable. Further, in Colt it is sometimes necessary to configure an IFU solely for the purpose of using a Skip Bus connection that passes over it. This seems to be a great waste of configuration time and resources. Finally, allowing streams to travel independently through Skip Bus segments as easily as they now pass through local connections would allow pages to be more easily fragmented across the mesh, since the Skip Bus segment passing over a given IFU would not need to be configured by the same stream that configured the

250

underlying IFU. This would allow a stream to "jump over" an intervening page in the mesh in order to act as a bridge between disjoint parts of another page. This system would need to be designed so that configuration information in the stream header would be latched at each node of the Skip Bus at each clock cycle; however, after configuration the operands in the data section of the stream could traverse multiple Skip Bus segments within a single clock pulse. The implementation of this last point would have the added effect of placing an added source of delays on the chip for execution path length equalization. Experience suggests that delays are frequently needed to equalize path delays, particularly on a large chip, and sacrificing FUs for this purpose can be wasteful. The latches used for configuration information by the Skip Bus could optionally be configured to latch the operand data as well, providing the needed delays.

7.1.11 IFU Routing Philosophy

The routing mechanism used to deliver operands from a local connection or Skip Bus segment to one of the FU input registers is very "IFU centric." This means that configuration bits are stored within the FU that control the IFU multiplexers which are used to route operands from any of eight different sources to the input registers. Indeed, it is possible for a stream to configure the IFU to accept operands from an input source that it has not even configured. Although this capability may occasionally be useful, it would be better to allow the stream path to entirely dictate the operand source so that data operands would exactly follow the path of the stream header. The IFU already partially supports this concept using the prioritization circuitry that is activated when a stream header arrives at an IFU from one of the four local connections. In Colt, after the stream header ends, this prioritization circuitry is switched off and the data

source is switched to the directions given by the configuration information. However, it would be more efficient to continue to use the direction defined by the stream header and do away with the configuration bits used to set the operand sources altogether. For one reason, the decoders that are currently used to switch between input sources could be replaced by smaller circuitry that mimics the IFU configuration prioritization circuitry. Mutually exclusive control signals can be easily derived from this system to directly drive the enable lines of the pass transistors with no need for an intervening decoder. This improvement would also greatly reduce the number of configuration bits needed to configure an IFU, making the design more efficient in terms of configuration time and the number of flip flops needed to store configuration information. Most importantly, it would reduce the number of bits that need to be moved across the precious I/O pins; the most valuable resource on the chip.

Of course, this change would necessitate a redesign of the path of configuration data through the IFU. It should be redesigned so that two separate streams can pass through the IFU simultaneously: one entering through the *Left Input Register* and the other entering through the *Right Input Register*. This would need to be done so that the data could truly follow the exact path taken by the stream header information through the chip. The normal configuration information for the FU could be taken from one of the streams, while the other would merely pass through. It would be advantageous to be able to take the configuration information for the FU from either of the input streams since it is often the case that one or the other of the input registers is not used and so no stream would need to be directed to it. Further, either input stream should be able to exit out of either the Bus Output or the Aux Output of the FU for the same

252

reason: different operations require the stream to enter through different inputs and the output that the stream needs to exit is just as variable.

This improvement would work well for the operand paths, which are wide enough to carry a configuration stream, but the connections used for the flags are another matter. Configuration data could be sent down these paths in the same manner; however, the single bit data path provided would be awkward to work into the rest of the chip which is designed around a 16-bit data path. Most likely it would be necessary to route the flags using the existing scheme, although further thought as to how this could be made more like this improved data routing scheme is warranted.

7.1.12 FU Complexity

Since the FU occupies only 25.4% of the area of the IFU, it may be reasonable to remove some of the connectivity of the IFU and instead place two or three IFUs in the same space, each with reduced connectivity. IFUs would then need to be sacrificed for routing, etc., but a potential for higher computational density would be gained. Alternatively, more computational power could be included within the FU, such as including a multiplier or multiple ALUs, shifters, more conditional logic, etc., as dictated by more practical experience with algorithm mapping.

7.1.13 Multiplier Improvements

In the current Colt design, the multiplier only performs unsigned multiplication. It would be useful to design the next generation multiplier with the ability to performed signed multiplication and also integer division. These would greatly assist in implementing

mathematical operations that, at this point, must be at least partially implemented using FUs. The hardware added to the multiplier to implement signed multiplication would not be significant and it is my belief that the addition of division capabilities would have minimal impact as well.

That the layout of the multiplier that is used in Colt is not optimal and it could be made significantly smaller. If the size of the multiplier could be halved, which seems likely, then it may become feasible to include a multiplier in every FU, although this may be extreme. It is at least likely that several multipliers will be included on the Stallion. I would recommend that at least four be included on the larger chip and possibly more. Also, the decision to directly connect the multiplier to the crossbar on the Colt has proven to be a good one in practice when mapping algorithms. The exact placement of the multiplier within the stream of execution is a moot point with the current implementation because of the flexibility that the direct crossbar connection allows, whereas if it were embedded in the mesh the paths of execution would have to be bent to meet at the correct location, making the mapping problem much more difficult in the limited environment of the mesh.

7.1.14 Barrel Shifter Inputs

Currently, when shifting to the left by more than one bit, the barrel shifter takes the bits shifted into the lower positions from the upper bits of the *Right Input Register*. It would be convenient to have two modes for this function, one would operate in the current manner and the other would always shift zeroes into the lower bits of the result. This would make it easier to efficiently implement some functions requiring shifts, such as multiplication using the shift and

254

add technique discussed, because the programmer would not need to guarantee that the upper bits of the *Right Input Register* were zeroes.

7.1.15 Barrel Shifter Data Path
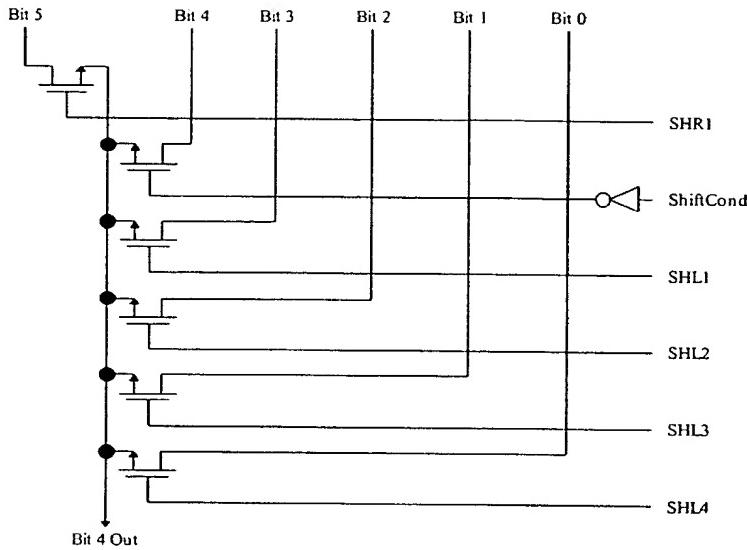
Figure 7.1- Alternate Barrel Shifter Design.

The barrel shifter design shown above would be an improvement to the one that was used in the Colt. Note that the signal only goes through one NFET pass transistor instead of the two that are currently traversed. The key to this design is that the *ShiftCond* signal is used as an enable for the 3:5 decoder that drives the shift select lines. When *ShiftCond* is 0, the decoder is disabled and none of the shifted values are enabled; only the original version of bit 4 passes through the device. Naturally, this same structure would be used for all the other bit positions. This design does have the drawback of requiring the decoder select lines to have single clock cycle response time, which is something that the previous design did not require. It may also be

255

beneficial to construct a true barrel shifter that takes a variable shift value as input and can shift by an arbitrary number of bits between 0 and 7, for example. The size of such a unit would no doubt be larger and this tradeoff would have to be weighed accordingly.

7.1.16 Conditional Unit Modes

Originally, the *Conditional Unit* had two modes of functionality: the current mode of selecting between the output of the ALU and the *Right Input Register* and also a mode that allowed the programmer to select between the output of the ALU and zero. The VLSI implementation of this addition is extremely small in terms of the data path and it was only removed because the configuration bit needed to select between the two functions was needed for something else. However, cases have arisen, even when mapping the few examples given in this work, that could have benefited from this function, and the cost is only a single configuration bit.

7.1.17 Flag Clocking

The algorithm mapping problem would be greatly reduced in many circumstances if it were possible to optionally latch any given flag at various times. Currently, the carry flag (*FCIn*) is always latched before it is fed into the FU, just as a full 16-bit operand would be. In some circumstances the timing of the algorithm makes it desirable to latch other flags before arriving at the final destination. In the Colt this requires either a redesign of the execution path or an FU dedicated to the task. It would be much more efficient to allow latching of an flag within the FU. The latching function should be optional because there are also occasions when the timing dicates that no latching occur.

7.1.18 Generalized Flags
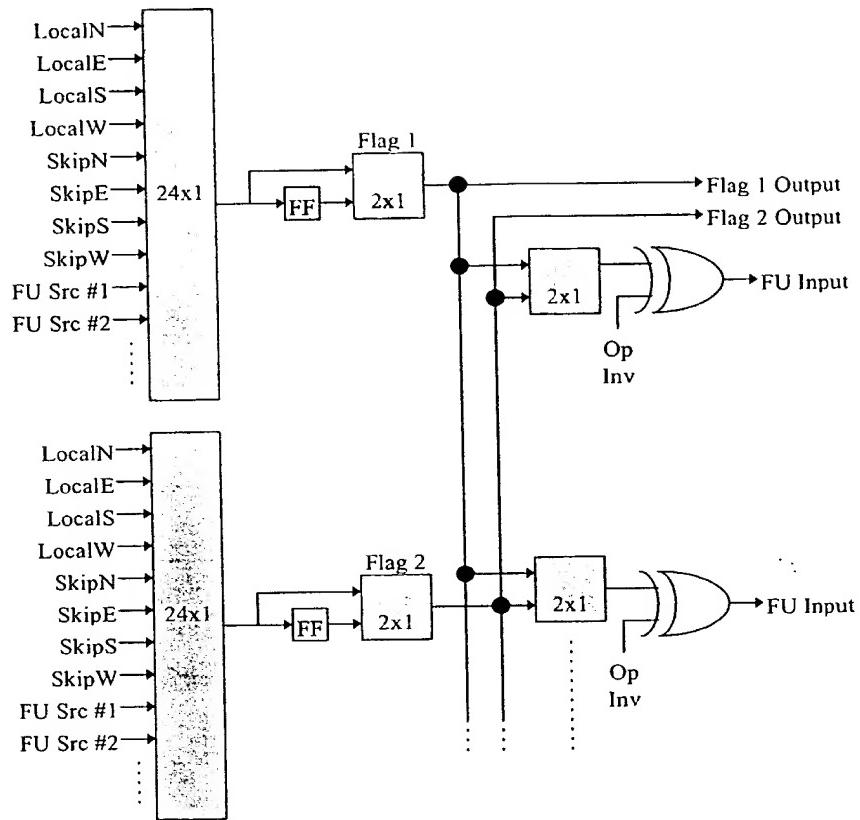
Figure 7.2 - Generalized Flag Concept.

The flag concept should be expanded so that instead of three special purpose flags, a set

of two or three generalized flags would be used within the FU. Any of these flags could be used

to control any function within the FU and any flag could use any input source that is available to

any other. Naturally, with such a design it would not be necessary to allow more than one flag to

connect to a given input source at the same time, which might allow some reduction in the

complexity of implementing this improvement. In the Colt implementation, the *FN* and *FNOut*

flags receive heavy use while the *FS* family of flags is rarely used. This addition would be very small and would alleviate many algorithm mapping problems.

A suggested architecture for the flags is shown in Figure 7.2. Here there are two generalized flags: *Flag 1* and *Flag 2*. There are two input multiplexers to select from either one of the flag inputs that is routed to this FU, or one of the signals that is currently used as an input to the *FNOut* multiplexer, or any additional flag outputs from the FU that are not currently routed to that multiplexer, but have proven useful. The output of the 24x1 multiplexers then have independently selectable delay flip flops before being used as the final form of the flags. These flags are routed out of the FU as the flag outputs and are also routed to each flag input found within the FU. At each FU flag input, either of the two generalized flags can be selected and an optional inversion is also available.

7.1.19 Arbitrary Flag Logic Functions

There are occasions when programming the FU where it would be useful to be able to define arbitrary functions of the flag variables and use those functions to control the various units of the FU. It would be a small addition to the FU and the configuration bits to allow the programmer to define one or two arbitrary functions of two or three flag inputs to generate value(s) that could be selected as inputs for use by the generalized flags. These could be used as inputs to the 24x1 multiplexers shown in Figure 7.2. The input sources for computing these flag(s) would need to be considered since making them inputs to the 24x1 multiplexers would effectively reduce the number of flag inputs to 1. The function(s) may need to be designed with a third independent set of multiplexers.

7.1.20 Data Port Loop Back Mode

It would be relatively easy and of some use to allow a data port to be configured from inside the chip and then have the stream return to the crossbar instead of exiting the chip. This would allow a data port to be programmed as an input, for example, without external intervention. This function does violate the pure sense of stream programming; however, it may have practical value in situations where a full blown stream controller at an input data port isn't necessary. For example, this would allow a data port to be directly connected to the output of an A/D converter with no intervening control logic.

7.1.21 Data Port Control Pin Timing

It may be possible to redesign the data port state machine so that the flip flops for the Program, Write, Transmit and Receive pins would not be necessary. Removing these flip flops would allow faster control reactions; such as when a data port is programmed from the outside to be an input. In that situation, the Write pin may incorrectly float high for a clock pulse. The Write pin isn't being driven from the end of the stream header until the first data word because the flip-flop at the output of the data port state machine prevents the change in the driven value from propagating to the pin until the next clock pulse. The designer would need be careful that no oscillatory transitions were introduced where the state machine uses the control pin as an input and continually drives it to the inverted sense.

7.1.22 Data Port Operand Counters

A counter could be included in the data port so that Loop Mode could be programmed to allow a variable number of valid operands into the chip. If this were done the entire loop could be filled with data so that better pipeline utilization could be achieved, rather than having just one data set in the pipe at a time. Of course, to make this work the flow control would have to be such that there would always be data available to the chip whenever it was requested by the chip. If this was not done, the gaps introduced into the stream by invalid data would cause collisions as the operands moved around the loop since operand spacing would then be critical. Also, the execution time of the loop would need to be guaranteed in some way so that the order that the operands went into the loop would be preserved. For these reasons it may not be a great addition.

7.1.23 Data Port Initialization

Six new global control lines could be run between each of the data ports, much as the SyncBus does in the Colt. Each port would be assigned one of these signals as an input. When configuring a data port six configuration bits could be used to indicate what other data ports would be used with the same page. These bits could be written to the new control signals as a sign that each of those ports needs to be reconfigured before they can start reading or writing normal data. This would prevent the problems of initially synchronizing data after configuration discussed in Section A.4.3.6. It would also provide a solution to the short stream header problem in Loop Mode that is discussed in the same section.

7.2 Known Colt Bugs

The IFU state machine discussed in the detailed design chapter suffers from at least one bug. The purpose of the device is to latch the direction that stream configuration information is originating from so that if another stream attempts to program the FU while it is processing configuration information from the first stream header the second stream will be ignored. This is especially important when using broadcast programming when a given IFU sends the remainder of the stream header in multiple directions. In such a case, it is possible for one of the streams leaving the IFU to wrap around the edge of the mesh and come back to the originating IFU from a different side; apparently as a new stream header. The ProgramIn line is generated by the IFU state machine and it goes to the FU to flag programming information. Currently, this signal depends on the logical OR of the latched values for the four signals: ProgramN, ProgramE, ProgramS and ProgramW. New values for these signals are not latched until all four of the raw input versions of these signals go back to a logic 0. Consider then the case where a stream to goes through an FU, branches, wraps around the mesh and hits the same FU again. The FU will be in programming mode and will stay locked onto the correct stream source. When the end of the stream header gets to the IFU and data begins the FU should enter data mode; however, it won't because the end of the stream header won't have gone through the branch and yet and so the FU will still be getting configuration information from the second side. Because of this any data immediately following the stream header will be passed through as configuration information and effectively lost. This is an odd case and it could be prevented through proper manipulation of the stream data, but the design should be changed for the next revision.

7.3 Future Research Directions

There are many avenues to pursue for future research, both software and hardware. Obviously, future hardware that needs to be considered includes the development and design of the Stallion processor. There are many suggested improvements to the Colt architecture given here and the full implementation of these would require an extensive redesign of the existing components. More radical changes could also be considered, such as an improved scheme for handling data dependent calculations or an entirely different computational scheme within the Functional Units.

Another piece of hardware to be developed is the Stream Controller. There are many approaches that could be taken with this. The possibilities range from a simple queuing system all the way to an object oriented stream mapping controller that could dynamically map a given stream using various implementation strategies to available resources, while simultaneously coordinating with other Stream Controllers in a distributed fashion (Section 3.3.4.4). Other possible Stream Controller functions could include dynamic expansion and contraction of the hardware implementation of the stream as the computational needs of the system change, new jobs start or old jobs end (Section 3.3.4.3). The possibilities presented by this unit are many and exciting.

From the software point of view, there is much work to be done in automating the algorithm mapping process. Hand configuring the Colt to perform an given algorithm, or piece of an algorithm, can be a complex task. Considerations must be given to many factors including: resource availability, computational abilities of the various units, routing constraints between

262

units, execution path length equalization, packing density and other factors. An automated system for breaking an algorithm down into constituent computations that can be performed by single FUs and then mapping the FUs onto the available resources would be a complex, but invaluable, tool. Even a tool that could be used to store existing hand-configured applications in a library could be useful. Such a tool would be especially valuable if the library components could be built up in a hierarchical fashion, allowing the output of one to be connected to the input of another to create a new component would be useful. Automated generation of the stream(s) needed to configure the devices to perform these macro operations would complete the functionality of the library tool.

Applications for the Colt processor itself should also prove to be a fertile field for research. Obviously, there are many applications to which the Colt processor is well suited. Though the processing resources appear scant when implementing real algorithms, the run-time reconfiguration abilities of the Colt can be used to compensate. The first to be implemented will no doubt deal with the algorithms and computations involved with the transmission and reception of CDMA communications systems. However, there is an entire class of applications that center around matrix multiplication in conjunction with the multiplication scheme discussed in Section 4.6.3. Using this scheme, an entire 3x3 or even 4x4 matrix multiplication can be performed using the mesh. If a 3x3 system is considered, then three input ports and three output ports could be used to supply input and output vectors to the mesh. There are many possible applications of this technique, ranging from the construction of a three variable control system to coordinate transformations needed in computer graphics. A third field that should prove fruitful for investigation is the implementation of filters, curve fitting and other functions using spline

calculations that require only basic shift-and-add primitives as detailed by Ferrari [75,76,77,78,79].

7.4 Final Words

The design of the Colt CCM device has changed dramatically from the first vision and improved steadily throughout the design process. Indeed, the large number of suggestions for the implementation of future devices shows that it truly is a work in progress. By going through to final silicon design, Colt has established a data point in the continuum of development. This data point can be viewed as a new baseline for performance and design of CCM devices and computing as a whole. The contributions exemplified by the Colt CCM include a word-oriented approach, using larger functional units in an effort to gain efficiency in configuration time and execution speed.

In the face of the increasingly greater effective die sizes that the VLSI designer can exploit, the question remains of how best to utilize that area. Flynn [80] notes:

---

75 Wang, Shin-Ywan, Ferrari, Leonard A. and Silbermann, Martine J., "High-Speed Computation of Spline Functions and Applications," *International Journal of Imaging Systems and Technology*, Vol. 7, pp. 1-15, 1996.

76 Ferrari, Leonard A. and Sankar, P. V., "Minimum Complexity FIR Filters and Sparse Systolic Arrays," *IEEE Transactions on Computers*, Vol. 37, No. 6, pp. 760-764, June 1988.

77 Sankar, P. V., Silbermann, M. J. and Ferrari, L. A., "Curve and Surface Generation and Refinement Based on a High Speed Derivative Algorithm," *CVGIP: Graphical Models and Image Processing*, Vol. 56, No. 1, pp. 94-101, January 1994.

78 Ferrari, L. A., Silbermann, M. J. and Sankar, P. V., "Efficient Algorithms for the Implementation of General B-Splines," *CVGIP: Graphical Models and Image Processing*, Vol. 56, No. 1, pp. 102-105, January 1994.

"As technology scales to below 0.6μm, architects need to explore higher-issue superscalars and more than two multiprocessors on a single chip. The benefits of larger on-chip caches decline as these caches reach 128K (total) and beyond."

The Colt CCM and other devices based on Wormhole RTR demonstrate a form of scalable multiprocessor system that can usefully employ that additional silicon.

More importantly, Wormhole Run-Time Reconfiguration has been offered as a computing paradigm that is unique in the architectural field. By offering the advantages of distributed control, localized communications, tolerance of latency and pipelined parallel processing, Wormhole RTR provides a means of raising the computational limits of CCM devices. Further, Wormhole RTR can be used on a larger scale in heterogeneous computing environments to create vast farms of computational logic that can be harnessed to any task, all linked and controlled by streams of programming information and data. This is the larger vision of the work presented here and will, I believe, become the contribution that is most recognized in the future.

79 Pang, D., Ferrari, L. A. and Sankar, P. V., "B-Spline FIR Filters," *Circuits, Signals and Signal Processing*, Vol. 13, No. 1, pp. 31-64, 1993.
80 Flynn, Michael J., *Computer Architecture: Pipelined and Parallel Processor Design*, Boston, MA, Jones and Bartlett Publishers, Inc., 1995, pg. 717.
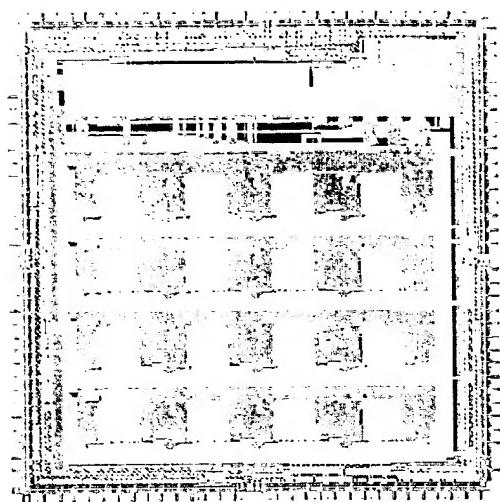
Figure 7.3 - The Colt CCM.
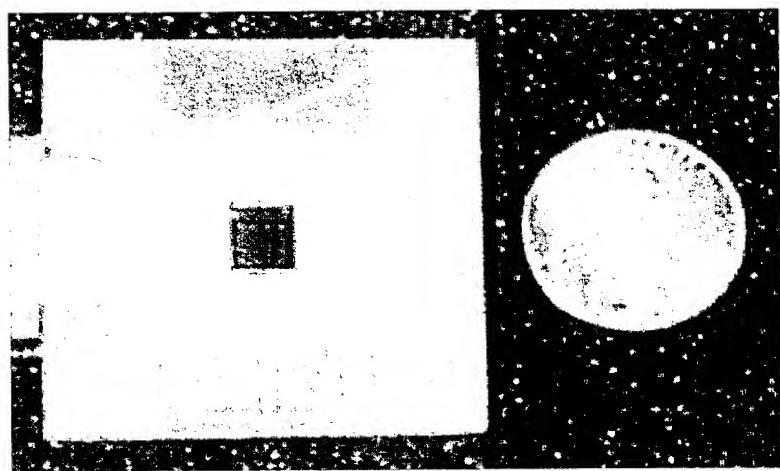
Figure 7.4 - The Colt CCM in Perspective.

266

Figure 7.5 – Ray Bittner and Peter Athanas.
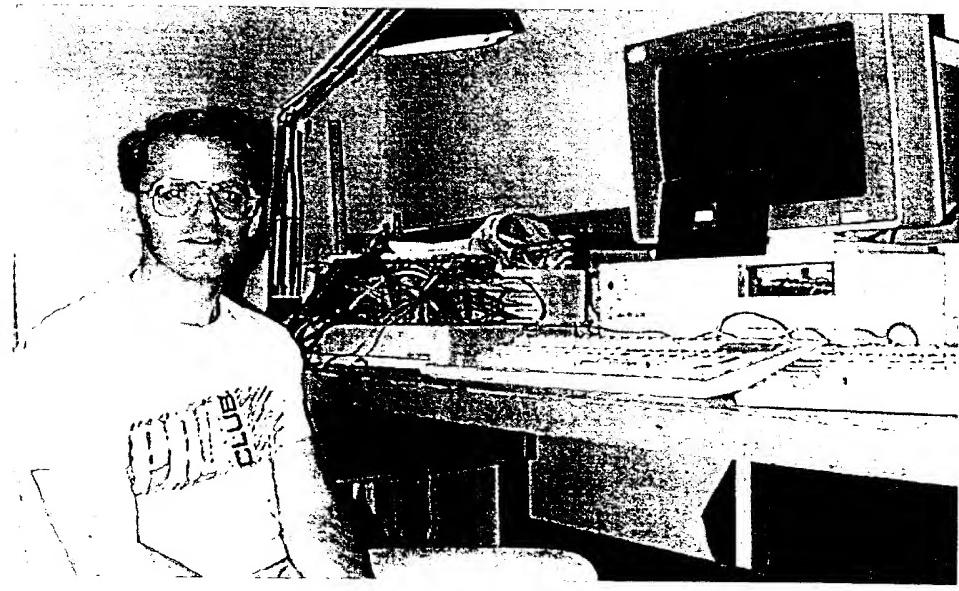
Figure 7.6 – Ray Bittner and the Test Apparatus.

267

A. Detailed Colt Design

Chapter 4 gave an introduction to the Colt architecture and basic functionality. Chapter 5 gave examples of how the architecture can be used to perform useful computations. This chapter gives an indepth account of the design and capabilities of all the units on the chip. Contained herein are all the key points needed to design and program the Colt CCM from high level design down to precise descriptions of the controlling state machines and the electrical design of the circuitry. Justifications are also given for many of the tradeoffs that were made during its creation. Because this chapter dissects the chip it can be tedious at times, and the Colt CCM can be appreciated from merely reading Chapters 4 & 5. However, those wishing to program the Colt or fully appraise its usefulness will need to sift through the details.

A.1 Design Philosophy

The major design philosophy behind the implementation of the Colt was "Local Complexity Over Global Routing." The increasingly small feature size of VLSI technology makes it possible to pack ever greater amounts of circuitry on a silicon die. While the amount of useable area is growing geometrically, the number of available I/O pins is growing only linearly. Hence, as described by Rent's Rule in Section 4.6.1, the pins available to the die become the scarcest resource on the chip. Further, the analogy extends inside the chip to routing resources. Routing long wires across the die is expensive in terms of area consumed by the wires

themselves, the power it takes to drive the wires in a reasonable amount of time, and the area consumed by the drivers. When possible it is preferable to use a greater amount of localized control circuitry to perform a given function than to use global routing to accomplish the same task if the same level of performance can be maintained. Fortunately, Wormhole RTR lends itself to a distributed control scheme, easing the achievement of this goal.

A.1.1 Pseudo-NMOS Circuitry

Another design tradeoff that was always considered was power consumption versus density of circuit implementation. Pseudo-NMOS circuits [81] were considered for logic implementation in many circumstances because such a circuit needs approximately half the number of transistors of the corresponding fully complementary circuit. The more complex the logic function, the greater the savings. However, these circuits also exhibit static power dissipation when the NFET pull down is in operation. In order to lower power dissipation, pseudo-NMOS circuits were avoided in most instances of the design; however, they were used occasionally for circuits that had few instances, but were rather complex, such as the Functional Unit state machine. Another advantage of pseudo-NMOS circuits that was exploited is the inherent ease of implementing bi-directional signals without concern for shorting out the circuitry. A pseudo-NMOS signal can be driven by a weak PFET transistor and pulled down by any number of driving NFET transistors with no harm to the circuitry. Safely accomplishing the same function using fully complementary circuitry requires negotiation for the bus signal and/or

extra control lines. This ability was used for the bi-directional control signals of the data port and also for the output busses of the crossbar.

A.1.2 High Strength Drivers
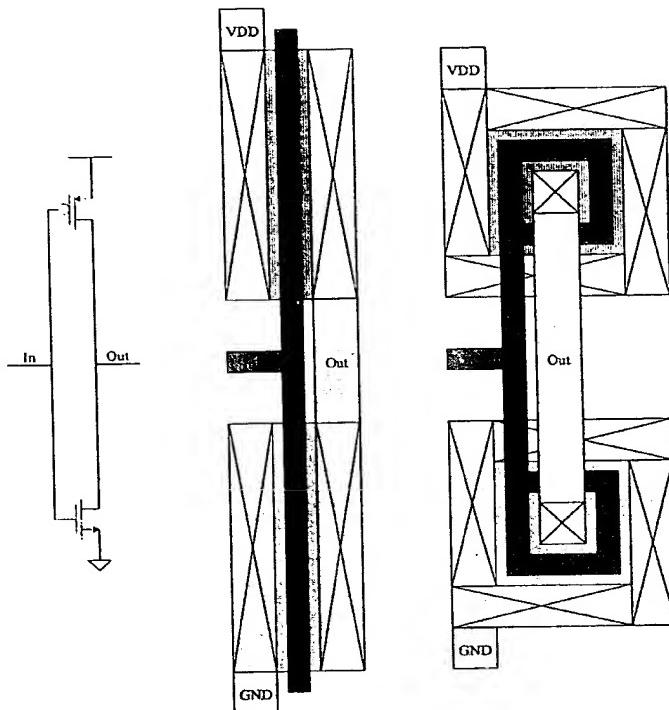
Figure A.1 - Alternative High Strength Driver Designs.

Figure A.1 shows two alternative CMOS high strength driver designs with similar strengths (W/L = 12). On the left is a symbolic representation of the inverter circuit that these drivers implement. In the center is what might be termed a linear layout, with the output

81 Weste, Neil H. E. and Eshraghian, Kamran, *Principles of CMOS VLSI Design*, Reading, MA, Addison-Wesley, Second Edition, 1993, pp. 73-77.

capacitance of the driver growing linearly with the W/L ratio of the transistors. The layout on the right is referred to as a donut in the literature [82] and was used extensively in the design. The advantage of this layout is that the output capacitance, derived from the source of the NFET and the drain of the PFET, is much smaller; hence the speed of operation of the circuit isn't impacted as badly by the driver as would be the case for the linear design. At the same time, the effective W/L of the doughnut is quite high, approximately 12 (W = 24λ, L=2λ), giving a high drive strength. Finally, since the capacitance on the power rails is higher for the doughnut, more charge is stored so that when the device changes state, more charge is immediately available for delivery to the load resulting in lower propagation delays. For these reasons, doughnut drivers were used exclusively throughout the design. When more drive strength was required multiple instances of these drivers were used.

A.1.3 Flip Flop Design
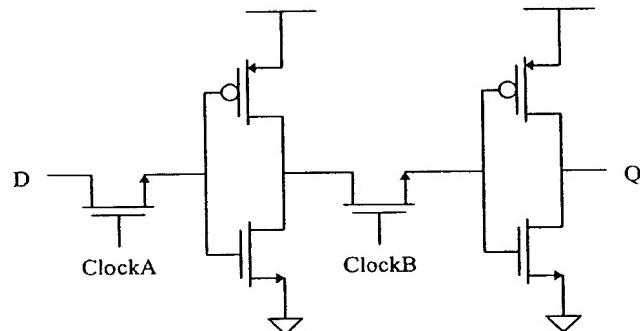
Figure A.2 - D Flip Flop Design.

---

82 Weste, Neil H. E. and Eshraghian, Kamran, *Principles of CMOS VLSI Design*, Reading, MA, Addison-Wesley, Second Edition, 1993, pp. 277-278.

271

The basic flip flop design used is shown in Figure A.2. This is a dual phase dynamic flip flop, storing charge representing the current state at the inputs of the two inverters. There are two clock signals needed to drive the device, called ClockA and ClockB, which must be non-overlapping, two phase signals. The generation of these signals is discussed in Section 4. Each of the clock signals drives an NFET pass transistor to gate charge through to an inverter stage. Unlike some other dynamic designs, this flip flop has active drive strength at the output terminal, eliminating many charge sharing problems. However, the NFET pass transistors can degrade the input signal when passing a logic 1. A logic 0, or 0 Volts, suffers no degradation of signal strength. Simulation showed that the initial voltage drop from 5 Volts through a series of NFET pass transistors was approximately 1.1 Volts for the first NFET and approximately 0.2 Volts for each successive NFET at normal speeds of operation. This implies that a logic 1 arriving at the first stage inverter inputs will be at most approximately 3.9 Volts, giving a Vgs of 3.9-5 = -1.1 volts for the inverter PFET device and hence it may no longer be in the cut off region. The NFET will not be in the cut off region either, as the Vgs for the NFET is then 3.9 Volts. Fortunately, the process gain for the 0.5 μm Hewlett Packard process used for the Colt is 3.8 times larger for the NFET as compared to a PFET device. This is within the design range of proportional drive strength needed to construct a pseudo-NMOS circuit. Thus, the NFET and PFET of the inverter stage were given the same geometries so that when a degraded logic 1 arrives at the input of the inverter the stage acts as a pseudo-NMOS circuit and produces the correct result. This mode of operation does have the disadvantage of producing static current dissipation on the order of 25 μA, but this was considered secondary to the area advantage of creating a six transistor flip flop.

272

Though the flip flop design shown is inherently dynamic in nature, a pseudo-static design can be derived from it by using two pass transistors as a 2x1 multiplexer. One input of the multiplexer can be used to reload the last value of the flip flop and the other can be used to load a new state when appropriate. As long as the clock continues to run, the value stored in the flip flop will not be lost. The multiplexer control signal then effectively becomes a load signal. Further, if testability were more of an issue, a third such transistor could be added for use as a data path in a scan path type design. In fact, pass transistor logic such as this was used throughout the design, particularly in the data path, to lower the silicon area used and produce faster designs.

A.1.4 Clock Driver Design
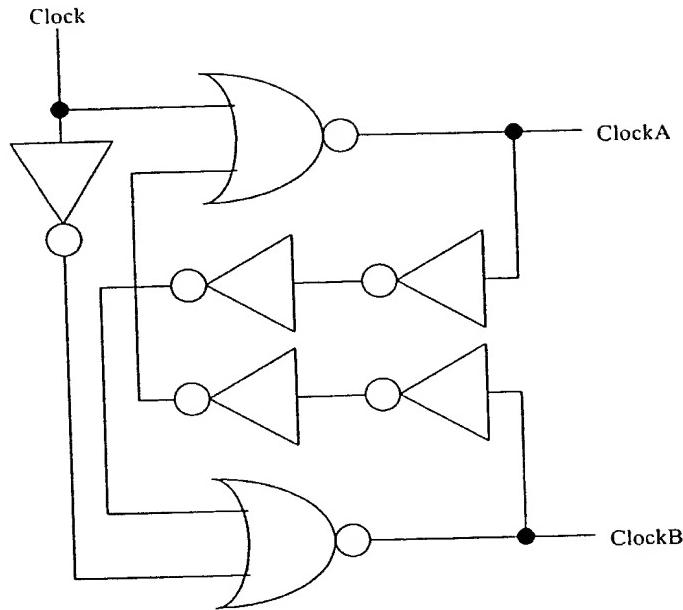
Figure A.3 - Clock Generator Circuit.

The two phase clocking scheme required for the flip flops is critical to basic operation of the Colt. It is imperative that the two phases of the clock do not overlap. This precluded the use of two signals from off-chip pins as the twin clock signals. If the two signals were generated off-chip, it would be difficult to control the skew between the two at high speeds because of excessive rise and fall times for the large distribution network. For this reason is was decided that a signal clock signal would be supplied from off-chip and that this would be used to generate the twin clocks that drive the flip flops. An asynchronous state machine of some type was needed that triggered off of the normal input clock signal to generate the two phases. Several designs were considered before finally settling on the one shown in Figure A.3.
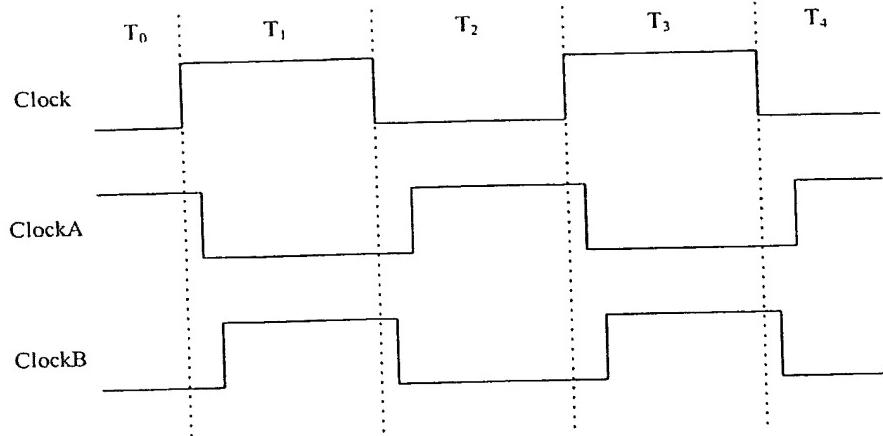
Figure A.4 - Clock Generator Timing.

Figure A.4 shows a timing diagram of circuit operation. At time T_0, the Clock input is low, allowing ClockA to be a 1 and forcing ClockB to be a 0. At the beginning of T_1, Clock goes to a 1. This forces ClockA to 0 immediately and also drops the lower input to the lower NOR gate to a 0. The upper input to the lower NOR gate is still a 1 though, and will continue to be a 1 until the ClockA signal voltage has completely dropped to a logic 0. This provides a feedback loop preventing ClockB from going high until ClockA has gone completely low. Once ClockA has gone low, the lower NOR gate will put out a logic 1 and pull the ClockB line high. Note that the feedback mechanism allows the clock generator circuit to "monitor" the actual voltage on the ClockA line so that the amount of time between the fall of ClockA and the start of the rise of ClockB is dependent on the load placed on the ClockA line. This guarantees the separation of the two clock signals in all circumstances.

To complete the cycle, the Clock line goes low at the beginning of T_2. This forces ClockB to go low. Again, the feedback mechanism prevents the lower input of the upper NOR

gate from going low until the ClockB line has gone to a sufficiently low voltage. Once ClockB has reached a logic 0, then ClockA will be pulled up to a logic 1 by the upper NOR gate. Hence the cycle is complete and feedback aids the separation of the two clock signals on both the rising and falling edges of the Clock signal.

To further guarantee the separation of the two clock phases, a number of physical layout considerations are taken into account. First, the two inverters are placed in the feedback path to slow the propagation of the ClockA or ClockB signal back through to the NOR gate inputs. The output response of these inverters has been skewed. The first inverter along the feedback path has been skewed so that an abnormally low voltage is required to generate a transition to a logic 1 output. This is accomplished by making the NFET strength proportionally larger than the PFET strength. Due to the fact that the NFET process gain is 3.9 times stronger than that of a PFET, this is quite easy to accomplish. The effect is enhanced however, by altering the geometries of the two transistors so that the W/L of the NFET is significantly larger than that of the PFET. This is necessary because the object is to make the input switching voltage of the inverter much lower than that of any circuitry that the NOR gate may be driving, thus helping to guarantee the separation of the two clocks. The second stage inverter along the feedback path is also skewed, but in the opposite direction since the object at this point is to make the output transition from 1 to 0 slow. The second stage inverter is skewed with the W/L of the PFET proportionally larger than the W/L of the NFET in normal gates found throughout the circuit. This will raise the input voltage required to forced the output of the inverter to a logic 0. Hence, the two stages of inverters work together to ensure that the two clock phases do not overlap. Finally, the outputs of the NOR gates themselves are naturally skewed because a NOR has two

276

PFETs in series, whereas there is only a single NFET driving the output. Hence, the W/L of the pullup is half that of the pull down transistor. Add to this the fact that the process gain of the NFET is twice that of a PFET with the same W/L ratio, and it becomes obvious that the rise time of the NOR gate will be slower than the fall time. All of these effects prevent overlap of the two clock phases.

Note that clock skew between the two phases could also cause a great many problems in the circuit if it were not taken into account. To prevent skew, a separate clock generator circuit was used locally with a given set of flip flops. The general rule of thumb used was one clock generator to every 17 flip flops, which is the number of flip flops in a normal Colt register. The clock generator circuitry is rather small, approximately ¼ the size of such a register, so for the sake of safety this perhaps overly cautious measure was taken.

A.1.5 Stream Conventions

While designing the control for the various units in the system, a number of conventions were used with regard to how various situations were handled as streams passed through the system. First, a stream is defined as consisting of two sections: header and body. The header section, also called programming data, is delineated by the Program signal being set to a logic 1 throughout. The body section, also called operand data or simply data, consists of all words following the programming section for which the Program signal is a logic 0. If Program returns to a logic 1 again, a new stream has started.

The valid bit is used for both the programming and data sections of the stream. In both cases, the intention is that if the valid bit is a logic 1, then the word should be treated as useful

information for the system. If the valid bit is a logic 0, then the associated word should be treated as garbage and will be ignored by the system.

The stream header is further subdivided into individual packets, each of which is intended to program a single unit along the path of the stream. The packet for the next unit to be programmed is always at the head of the stream. A unit that is being programmed will strip the lead packet off of the front of the stream and will not forward that information. When a unit has stripped all information for its packet from the stream, the remainder of the stream will be forwarded to the next unit along the stream path. The packets can be variable lengths and have different formats for different types of units. The only standardization between packets is that bit 15 of all configuration information is used as a flag to indicate the beginning of a new packet. The first word within a packet must have bit 15 set. Bit 15 must have a logic 0 value for all valid words following the first that are part of a multi-word packet. Note that a design tradeoff was made here. Bit 15 could have been used for normal configuration information rather than for use as a flag bit. However, if this were done, all units requiring more than one configuration word would have required a counter of some sort to determine when the end of the packet had been reached. All units on the Colt require only one configuration word, except for the Functional Unit, which requires eight. In hindsight it may have been better to design a counting mechanism for the FUs and use bit 15 for other purposes as this would have better fit with the local complexity over global routing theme. Bits 5 down to 0 of the first word of a packet are defined to be the address of the unit that the packet is intended to program. Bits 14 down to 6 of the first word in a packet are undefined and may be used for unit programming.

Table A.1 - Colt Unit Addresses

Unit	Address
Crossbar Output - Mesh Column 1 Local	1
Crossbar Output - Mesh Column 1 Skip	2
Crossbar Output - Mesh Column 2 Local	3
Crossbar Output - Mesh Column 2 Skip	4
Crossbar Output - Mesh Column 3 Local	5
Crossbar Output - Mesh Column 3 Skip	6
Crossbar Output - Mesh Column 4 Local	7
Crossbar Output - Mesh Column 4 Skip	8
Crossbar Output - Mult High Word Output	9
Crossbar Output - Mult Low Word Output	10
Crossbar Output - Data Ports 1 through 6	11-16
Data Port 1-6	24-29
FU (Upper Left to Lower Right By Row)	32-47

Table A.1 gives the assigned addresses for each unit and crossbar output on the chip. In addition, the binary unit address 111111 is defined to be the broadcast address and is treated differently by each unit type. The general intention is that any unit receiving a packet beginning with this address should respond as if the exact address for that unit had been specified. There are a number of exceptions to this rule as some possible situations do not make sense and so were disallowed in the hardware. These are noted case by case. Further, the binary address 000000 is not used on the Colt. This is defined to be a NOP address that is guaranteed to illicit no response from any unit. This address can be used as a flag of some type or as valid filler in the programming header if needed.

A Program Active state was included in the controlling state machines of all units. A unit is in this state whenever it has been programmed, but the end of the programming header has not

yet been reached. When a unit is in this state it will not respond to any information in the programming header, not even a packet containing the correct address for that unit. The intention of this feature is to allow several units having the same address to be used in the same stream path without interfering with one another. If this feature were not included, then globally unique unit addresses would have to be used throughout the stream driven system; even between different Colt chips. But, because of this feature, a unit can be programmed using the correct address, and then all further programming information is simply forwarded on to the rest of the system. If a packet destined for the corresponding unit in another chip is contained in the programming header, it will be passed through the system intact; eventually programming a unit on another Colt chip in some other remote part of the system.

A.1.6 Address Comparators
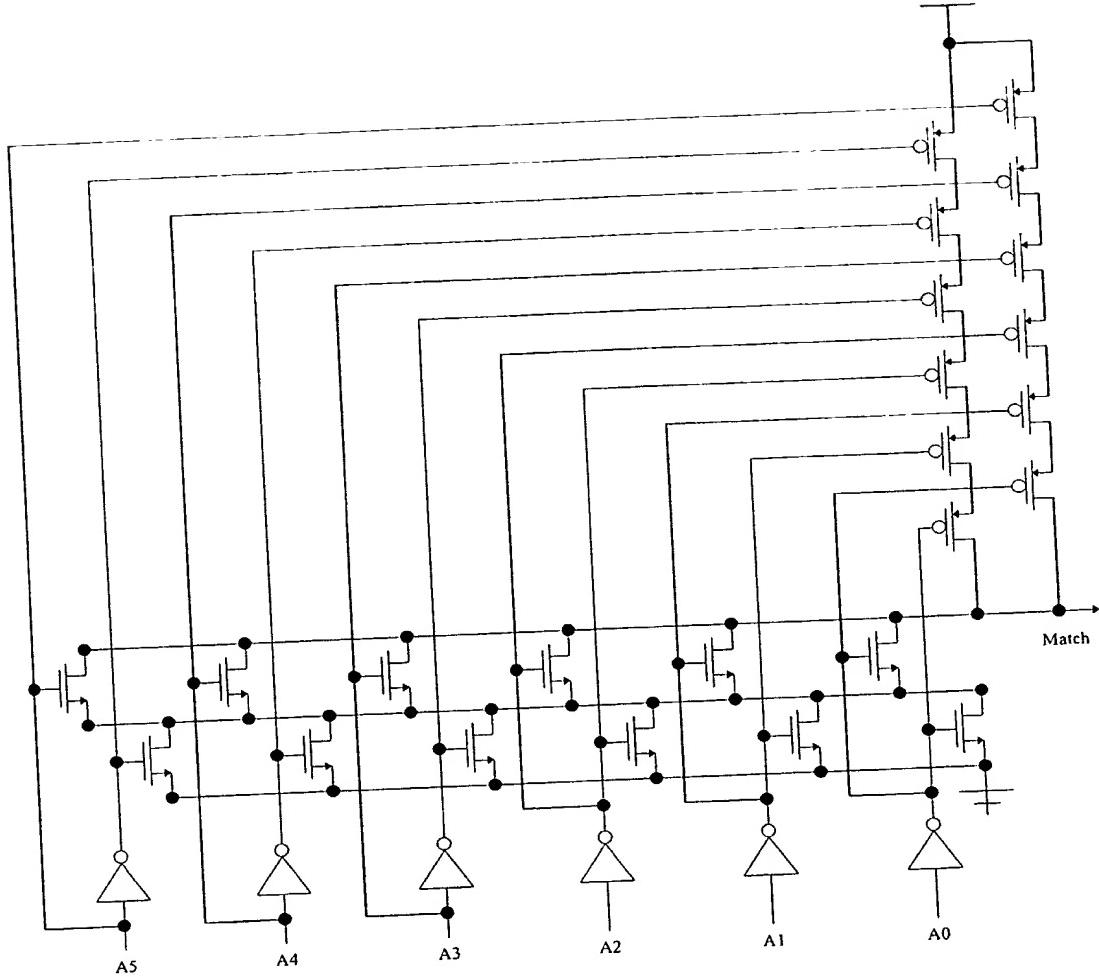
Figure A.5 - Address Comparator Circuit.

An example of the circuit used for the address comparators is shown in Figure A.5. Note that the *Match* output will be logic 1 for either address 7 (000111) or for the broadcast address 111111. For all other addresses the output will be logic 0. A smaller circuit could have been built using a pseudo-NMOS design, but it was decided that the savings in current drain of going

with a fully complementary version was worth the loss of area. The regularity of this design also eases layout since only one or two wires per bit need to be changed to create a comparator for a different address.

A.2 Multiplier

The design of the multiplier was handled by another researcher at Virginia Tech and the details of the design will be documented by him. For purposes of use, it is sufficient to say here that it is a pipelined 16-bit x 16-bit unsigned multiplier producing a 32-bit result after a latency of two clock cycles. The multiplier resides off of output addresses 9 and 10 from the crossbar. The 32-bit output is split into high and low 16-bit words, each of which has a separate connection back into the crossbar. When programming, the stream entering from crossbar output address 9 will exit through the high word of the result, and the stream entering from crossbar output address 10 will exit through the low word of the result. All stream header information is passed through the device along these paths with no changes and without stripping any programming packets since the function of the multiplier is not programmable. If only one input is accepting programming information, then the data from the opposite output is flagged as being invalid. When neither input is accepting programming information, the valid bits of the two inputs are logically ANDed together to form the valid bit that is used with both the high and low word of the result.

A.3 Crossbar

To support the stream processing concept, the crossbar was constructed with a distributed feed forward control structure so that an entering stream could select an output path and have the rest of the stream flow along that chosen path. Several streams could be directed through the crossbar simultaneously; each with a different destination. Streams cannot be halted within the chip, giving rise to the requirment that all routing "requests" be handled simultaneously. This is a difficult requirement for a centralized controller to meet; whereas such feats are trivial for a distributed control system. Further, the local complexity over global control philosophy is better satisfied by the use of distributed control mechanism. For these reasons, a distributed control mechanism was developed.

A.3.1 Data Path

Section 4.6.2 shows the high level construction of the crossbar. The crossbar has 12 inputs, each of which is represented by a horizontal bus running across the width of the entire unit. 16 outputs are driven by vertical buses running down the full height of the crossbar. A stream entering a row uses the unique address for the column to select the output path for the stream. At the intersection of each row and column is an *XBarNode*, consisting of a register to latch the data coming in on the attached row and a state machine that is used to determine if the data from the row should be transmitted on that output column. Note that the vertical output buses are open drain so that if a programming error occurs and two or more *XBarNodes* drive the vertical bus simultaneously no short circuits will be created.

283
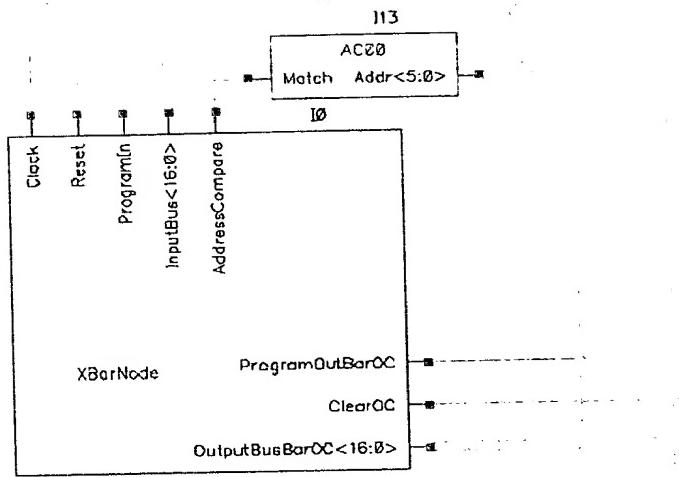
Figure A.6 – Crossbar Intersection.

Figure A.6 shows the *XBarNode* at the intersection of a row and column. The data

connections across the row are the 17-bit *InputBus* and its associated single bit *ProgramIn* line.

The *AC01* component is an address comparator that is used to test the first word in the first

packet in a programming header to determine if this column is being selected as an output for the

new stream. The outputs are *ProgramOutBarOC*, *OutputBusBarOC<16:0>* and *ClearOC*. The

first two are what would be expected, except that they are open drain drivers that are used to

prevent contention on the vertical output bus if a programming error occurs. Also, these are the

inverted forms of these signals because a savings in hardware was gained by allowing the natural

inversion of the open drain driver to be used on the output bus. These signals are recovered in their true forms at the bottom of the column.

The third output signal is *ClearOC*. Again, this is a open drain signal so that it can be simultaneously pulled low by multiple *XBarNodes* on this column in the event of a programming error. *ClearOC* is used as a broadcast alert to all *XBarNodes* on that column; signaling that during the next clock cycle a new stream will take control of the crossbar output. Upon seeing the assertion of this signal, all *XBarNodes* will reset and will discontinue transmitting data on that output column. During the following clock cycle, the *XBarNode* that asserted the *ClearOC* signal will begin transmitting on the output column.

A.3.2 State Machine
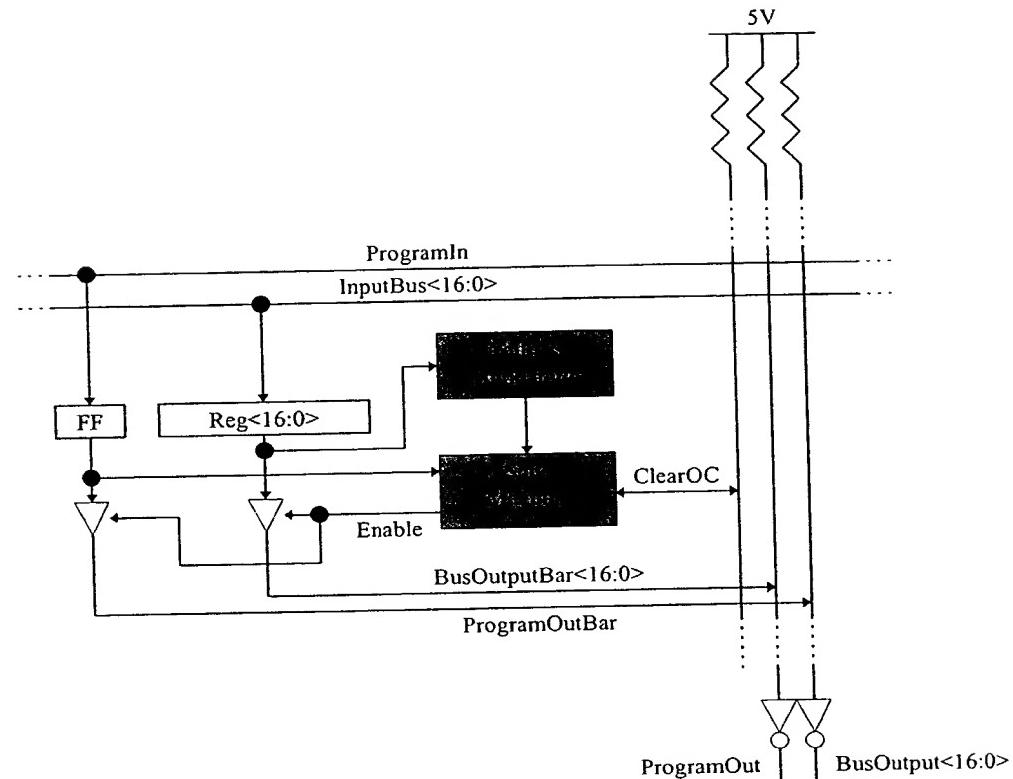
Figure A.7 – Detailed XBarNode Intersection.

A high level *XBarNode* internal schematic is shown in Figure A.7. This shows the flip

flop and register used to latch the input *Program* and *InputBus<16:0>* lines, respectively, as well

as the logic needed to control the switching of the input data to the output column. A Mealy state

machine determines when the switch should activate and, when active, enables the output to

drive the data onto the output column. The address comparator is driven by the output of the

register, not the input bus, thus forcing the use of a Mealy machine as the control circuit. If a

Mealy machine were not used, either an extra clock cycle would be required to program (turn on)

286

the switch, or the address comparator would need to be connected to the input bus line. If the address comparator were connected to the input bus line, the propagation delay time to the point of the input register would have to be shortened to allow the state machine to react. The critical propagation time is from the source to the input register since this distance may be relatively long, and; hence, it was desirable to allow as much time as possible to that point. On the other hand, it was anticipated that a long propagation time would be permissible on the output column of the crossbar; dictating that the address comparator should be connected to the output of the register.
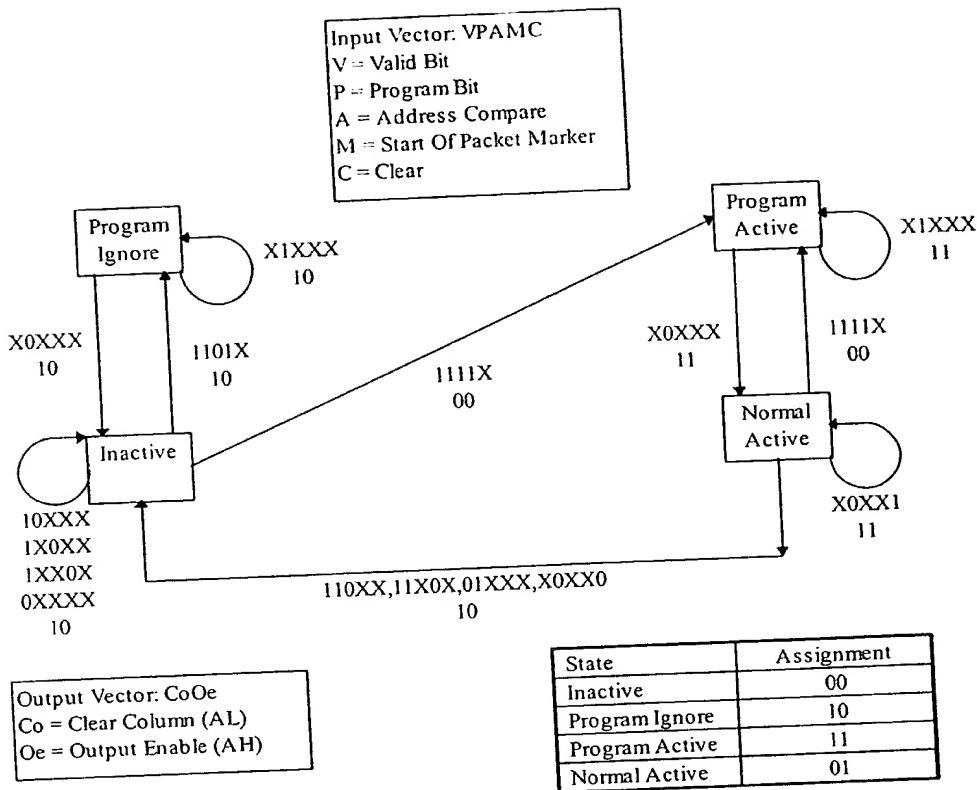
Figure A.8 - XBarNode State Transition Diagram.

Figure A.8 gives all key information about the state machine. The signal order of the 5-bit input vector used on all transitions is shown at the top of the figure. Likewise, the signal order of the 2-bit output vector is shown in the lower left corner and the state assignment used is shown in the lower right. The bit positions for the named bits are: valid bit (V) is *Reg<16>*, *Program Bit* (P) comes from the flip flop, *Address Compare* (A) is the output of the *Address Comparator, Start Of Packet Marker* (M) is *Reg<15>* and *Clear* is the *ClearOC* line.

Upon reset, the state machine is forced into the *Inactive* (00) state. In this state, the switch is effectively turned off and will not drive the output column. It remains in this state until

a programming word is latched into the buffers. If the state machine input word is 1111X, meaning that it is valid data, the programming bit is set, the correct address is present so that the address comparator signal is true and the start of packet marker is set, meaning that this is the first word of a programming packet, then the state machine transitions to the *Program Active* state. During the transition, the *ClearOC* signal is asserted by pulling it down to 0. This will signal all other switches on this column to turn off and stop echoing data onto the output lines.

The *Program Active* state is intended to allow the switch to remain on so that it can pass programming header data through to the rest of the chip using its associated output column. However, while in this state, the switch will not respond to any further programming from the stream. This is important because it allows other packets with the same address to pass through the switch without being stripped from the stream. This in turn allows multiple units with the same address to be connected in series as each can be programmed sequentially without previous units reprogramming themselves. Such capability would be used when directly connecting multiple Colt chips, for example. The state machine will remain in the *Program Active* state until a data word is latched into the register for which the *Program* bit is a 0, signaling the first word of the data section of the stream. This will cause the state machine to transition to the *Active* state.

Table A.2 - XBarNode State Equations.

Function	Equation
Q_1^+	$A\ M\ P\ V + M\ P\ Q_2'\ V + P\ Q_1$
Q_2^+	$\{Co\}' + \{Oe\}$
Co	$A' + M' + P' + Q_1 + V'$
Oe	$C\ Q_2\ P' + Q_1\ Q_2$

The switch is turned on in the Active state and it will continue to transmit data on the output column until one of three events occurs: it is reprogrammed, it is reset, or another *XBarNode* on the same column asserts the *ClearOC* signal. If a new stream begins (signified by programming information being latched into the registers again), the column driver may be reprogrammed to drive the column again. Note that care has been taken to ensure that if a new stream enters the input associated with the *XBarNode* and the new output address is not the same, the switch for that column will turn off and the state machine will return to the *Inactive* state. Also, throughout all transitions, the valid bit is monitored so that bad data will not cause a state transition. The next state equations for the *XBarNode* are shown in Table A.2.

It is important to recognize that the *Clear* signal C and the *Clear Column* signal Co are related in that *Clear* is the value of the *ClearOC* line depicted in Figure A.7, and *Clear Column* is the value that this *XBarNode* is attempting to drive onto the *ClearOC* line. These two may differ if this *XBarNode* the driving Co to a logic 1, but another node is driving its Co signal to a logic 0; the C signal would then be a logic 0. These two signals create a feedback loop from the state machine output to the state machine input. Also, since this is a Mealy machine, the feedback path can cause oscillations in the state machine output if care isn't taken to ensure that all state transitions leaving a state that has a transition driving *Clear Column* are mutually exclusive based on conditions other than the *Clear* input. For example, note the transition from *Inactive* to *Program Active*, which does drive the *Clear Column* signal to 0. Also, the *Active* state has one transition that does not depend on the *Clear* input but does drive it (1111X), and another that does depend on it, but does not drive it (X0XX0). These are mutually exclusive

290

based on the *Program Bit* input; however, and so oscillations are avoided. This was a good learning experience in state machine design as such oscillations were revealed in earlier versions of the logic by simulations and the root cause was eventually found.

A.3.3 Electrical Design

The key electrical aspects of the design all centered around the output columns. The first consideration was that two inputs could end up driving the same output column if two streams simultaneously programmed two different *XBarNodes* as outputs. This is considered to be programmer error; however, it was considered important to ensure that the chip cannot be harmed by any type of programmer misuse. Thus, the output column was designed using Pseudo-NMOS circuitry. The drawback of using this type of circuit is excessive static current drain and associated excessive power drain. Very weak PFET pull up transistors were used since this would result in less current drain when the signal was being pulled low. However, this also caused very slow rise times when the signal was intended to float high. A compromise was reached by assuming that a propagation time of 10 ns would be sufficient to drive any circuitry that was attached to the bottom of the crossbar output column. Using this assumption and the capacitive load on the output column, the PFETs were sized to drive the column within the remaining 10 ns of the 20 ns clock period.
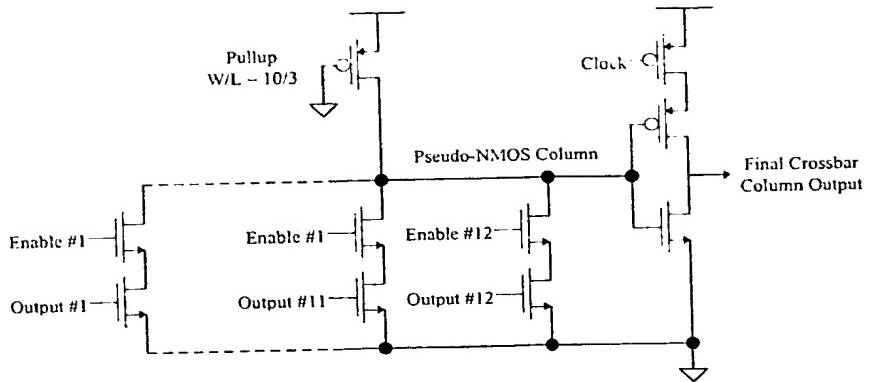
Figure A.9 - Crossbar Column Output Inverter.

This did decrease the power consumption of the column drivers dramatically; however, it had the interesting effect of greatly increasing the current demands of the inverters used to recover the true forms of the signals at the bottom of the crossbar columns. The reason for the new current spike was that the inputs to the inverters were rising very slowly, causing both the PFET and NFET of the inverter used on the output to be turned on for a much longer period of time as the input signal transitioned. After some careful thought, the circuit shown in Figure A.9 was used. The column drivers are shown as two minimum size NFETs in series, the top one is connected to the enable signal and the lower one is the actual value being driven, which is then, of course, inverted on the Pseudo-NMOS signal that travels down the vertical output column. The PFET pullup was sized to have a W/L ratio of 10/3 to satisfy a rise time of approximately 8 ns. In reality this was implemented as two PFETs, each with a W/L of 20/3. One was placed at each end of the column in an effort to reduce reflections of the signal down the long transmission line.

At the bottom of the column, the clocked output driver was placed to re-invert the Pseudo-NMOS signal to the true form. Note the clocked PFET in the final output driver. During the first half of the clock period the PFET will be turned off so that the output driver will not be able to drive a logic 1. However, using a 20 ns clock the rise time of the Pseudo-NMOS signal will take approximately 8 ns, thus it will take the entire first half of the clock period to charge up the input to the final driver in any event. Thus, the fact that the final driver will not function until the second half of the clock period is irrelevant from a propagation delay point of view. The relevance of the clocked PFET is that it prevents the normal PFET and NFET from both being turned on during the long signal transition caused by the slow rise time of the Pseudo-NMOS signal. Hence, the large current drain that had been caused by the long rise time can be neatly avoided and the power requirements of the crossbar drop significantly.

The only drawback to this solution is that the propagation delay from the crossbar output driver to the final destination must fit within the second half of the clock period (10 ns minimum). This would already have been the case even if the clocked PFET hadn't been inserted and it is really a consequence of weakening the Pseudo-NMOS pullup to save current drain at that point. In all situations, 10 ns is sufficient time for the signals to arrive at the destination as this is also the approximate amount of delay through the FU (being the worst case delay in the system); whose signals must also travel through much of the same routing resources.

293
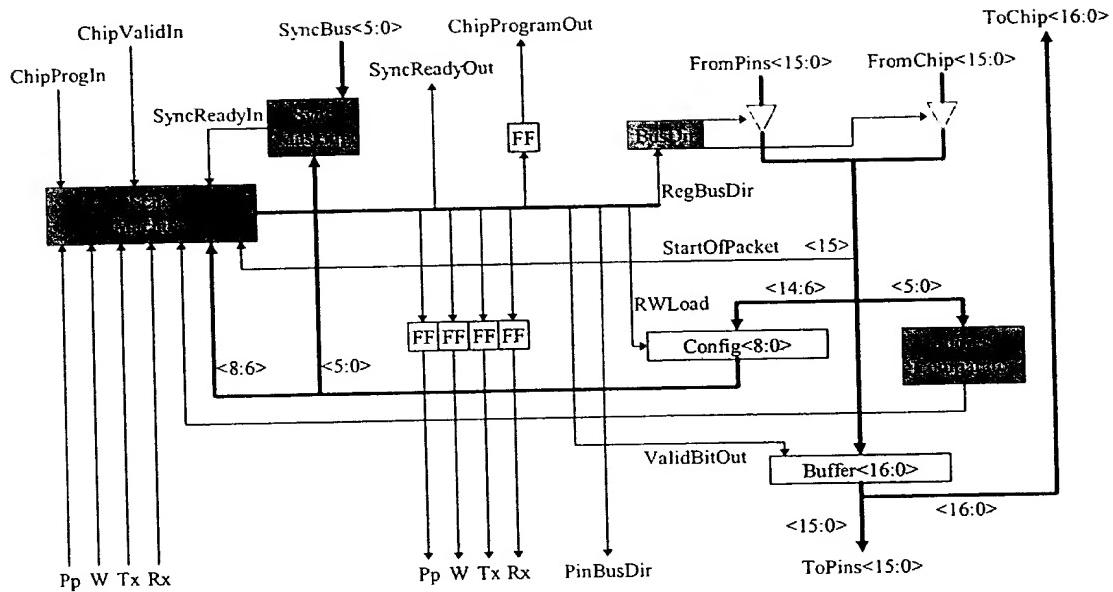
A.4 Data Port

Figure A.10 - Data Port Overview.

The bi-directional data ports were the greatest design challenge in the Colt. Each supports three modes of operation and is interconnected with chip resources, and to each other, to provide complex stream synchronization capabilities. A block diagratm of a data port is shown in Figure A.10. The left half of the diagram contains the control circuitry and the right half contains the data path. The state machine shown on the left receives 12 inputs and produces 10 outputs, not including the 2 inputs and 2 outputs required for the two state variables.

A.4.1 Data Path

Two data buses enter the unit: *FromPins<15:0>* and *FromChip<15:0>*. *FromChip<15:0>* are the 16 data lines coming from the crossbar to this data port. Note that the

valid bit, bit 16, for the data path from the crossbar has been separated and named *ChipValidIn*. *FromPins<15:0>* is the current state of the 16 data pads that are driven by this data port. The 16 data pads are bi-directional in the sense that they can be tri-stated and this bus always indicates their current state, regardless of the direction of data flow. The *PinBusDir* signal controls the tri-state properties of these pins and is 0 if they are to be used as inputs or 1 if they are to be used as outputs. The *ToPins<15:0>* bus contains the signals to be driven onto the 16 data pins if the port is configured to be an output. Likewise, the *ToChip<16:0>* bus carries the word currently latched by the data port back to the crossbar. Note that the valid bit is included in this vector as is normally the case. The *RegBusDir* signal multiplexes between the data from the pins and the data from the crossbar, controlling which is latched into the data port *Buffer* register. If *RegBusDir* is a 1, the data is latched from the crossbar (the port is being used as an output) and if *RegBusDir* is a 0, the data is latched from the pins (the port is being used as an input).

A.4.2 Flow Control

There are eight flow control signals associated with the pin side of the data port. There are input and output signals for *Program*, *Write*, *Transmit* and *Receive*. Each of these is implemented as an open drain pin on the Colt. Thus, it is possible for either the chip or an external source to pull the pin low at any time. The output signal from the data port is the value that the pin will have if no outside force interferes. Of course, since the pin is open drain, if the data port pulls the pin low it will have a logic 0 value regardless. The input signals to the data port from the pins is the actual value of the pad at the time. The difference between these signals and the outgoing signals is that it is possible for the value of the pad to be logic 0 even though

the corresponding signal leaving the data port is a logic 1. In this case, an external force has driven the pin to a logic zero.

The intent of the four flow control signals is as follows. The *Program* Pin is used to indicate when the header section of a stream (configuration information) is being transferred either into or out of the chip. The *Program* Pin will be low whenever configuration information is being transferred; it will be a logic 1 otherwise. The *Write* pin indicates the direction of data flow through the data port. A logic 1 on the *Write* pin indicates that data is flowing from the inside of the chip to the pins, this is a write, or output mode. A logic 0 on the *Write* pin indicates that data is flowing from the outside in, that is, from the pins to the internal circuitry of the chip, this is a read, or input mode. Note that the *Write* pin is almost always driven by the chip itself. The only exception to this is when a new stream (new programming information) is being sent into the chip. In this case, the external controller will pull both the *Program* and *Write* pins low, indicating that programming information should be read into the chip, and the configuration process will then commence.

The *Transmit* and *Receive* pins are used to implement stop and go flow control for data going through the port. *Transmit* is always driven by the data source and *Receive* is always driven by the data sink. Thus, if the port is configured for write mode, it will be responsible for driving the *Transmit* pin. If the chip allows *Transmit* to float high, then the information presented on the data pins is valid and should be latched by the external Stream Controller. If the chip pulls the *Transmit* pin low, then the information presented on the data pins is not valid and should not be latched by the external Stream Controller. When the port is in write mode the *Receive* pin should be driven by the external Stream Controller. This is a signal to the data

296

source, the data port in the case of a write, that the Stream Controller is nearing 100% capacity and that it can only accept a small amount of further data. If the Stream Controller lets the *Receive* pin float high then the data port should continue writing data freely, and no attempt should be made to slow the rate of the data being written from the port. If the Stream Controller pulls the *Receive* pin low, it means that only a small amount of data can be safely accepted from the port and that all attempts should be made to slow or stall the data stream going through the chip. It is important to note here that the pipeline through the Colt cannot be stalled, even in the event that a Stream Controller signals that it is full. Thus, even if the *Receive* pin is pulled low by the Stream Controller when the data port is writing, valid data may still be written from the port, as signaled by the *Transmit* pin. However, both Synchronization and Loop Modes support communications within the chip to signal the input data ports to stop accepting data. In this way, the amount of data that the full Stream Controller would have to accept can be limited to what is currently in the pipeline that exists on the Colt. Thus, it is important for a Stream Controller connected to an output port to implement the *Receive* pin as a nearly full signal so that it may accept the data remaining in the Colt pipeline.

When the data port is in read mode the situation is reversed. In this case, the *Transmit* pin is controlled by the external Stream Controller and is used to indicate when the data present on the pins is not valid (*Transmit* = 0) and should be flagged with the valid bit set to a 0 internally to the chip. If the *Transmit* pin is allowed to float high, the Stream Controller is indicating that the data is valid and that it should be flagged as such by the valid bit being set to a 1 internally to the chip. Likewise, the *Receive* pin is the signal from the data port to the stream controller that it could not accept the data value that was presented during the previous clock

297

cycle. If the data port allows the *Receive* pin to attain a logic 1, then the data issued during the previous clock cycle was accepted and a new value should be presented during this clock cycle. If the data port pulls the *Receive* pin low, the Stream controller should hold the value presented during the previous clock cycle on the data pins until a clock cycle occurs in which *Receive* is a logic 1. In contrast to the situation when the data port is writing, the response to a reading data port by the Stream Controller to the *Receive* pin going low must be almost instantaneous. This was required because in either Synchronization or Loop modes the data port may make an attempt to stall an incoming stream because either another input data port is no longer receiving valid data (*Transmit* = 0), or an output data port has gotten a nearly full signal (Receive = 0) from a Stream Controller. While stalled, the reading data port will inject data into the chip that has the valid bit set to 0, indicating that it is invalid. A data port in write mode will avoid writing invalid data to memory by pulling the *Transmit* pin low for any data word whose valid bit is 0. Thus, simple flow control can be implemented using the valid bit and the various modes of the data ports.
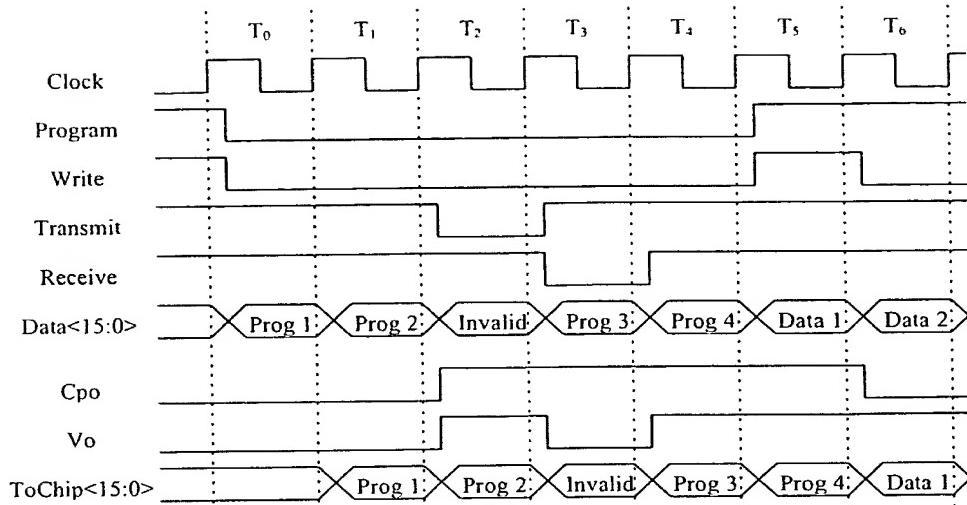
Figure A.11 - Programming a port as an input.

Figure A.11 shows an example timing diagram for the signaling sequence used when a Stream Controller injects a new stream into a data port and programs that data port to function as an input. *Cpo* is the *ChipProgramOut* signal, which is the program signal that is forwarded inside the chip. *Vo* is the valid bit that accompanies the *Cpo* signal and *ToChip<15:0>* is the actual 16-bit data word that is forwarded to the crossbar. Note that *Vo* becomes *ToChip<16>* before leaving the data port.

During T_0, the Stream Controller pulls both the *Program* and *Write* pins low indicating that programming mode has begun and the first configuration word is present on the data pins. Since the *Transmit* pin is high, the data is valid and it is latched by the data port. Because it is the first valid word of configuration information given to the data port, an address compare is performed and it is latched as the configuration word for the data port itself. It is not passed on internally to the chip as indicated by both *Cpo* and *Vo* being logic 0 during T_1. The second

299

programming word, called *Prog 2* is presented to the data port pins during T_1. This is valid configuration data and so it is latched by the data port *Buffer<16:0>* register. During T_2, *Prog 2* is forwarded to the rest of the chip through the *ToChip<15:0>* bus. It is flagged as configuration data because *Cpo* is a logic 1 and it is valid data because $Vo = 1$. At time T_2, an invalid data word is presented to the port pins, as indicated by *Transmit* being pulled low by the Stream Controller. This is latched into the *Buffer<16:0>* with *Buffer<16>*, the valid bit set to 0. Then at time T_3, the invalid configuration word is forwarded on to the rest of the chip. Configuration words *Prog 3* and *Prog 4* continue in similar fashion and programming ends during T_5, at which time the data section of the stream begins. To signal this, the Stream Controller allows *Program* and *Write* to float high, but *Write* is pulled low again by the data port because the port has been programmed to use input mode. Data words are then read into the chip and forwarded along the path defined by the configuration stream, using the same flow control protocol with the *Transmit* and *Receive* pins. Note that *Cpo* is low during T_6 because operand data is being forwarded into the chip, but the *Vo* bit continues to function normally.
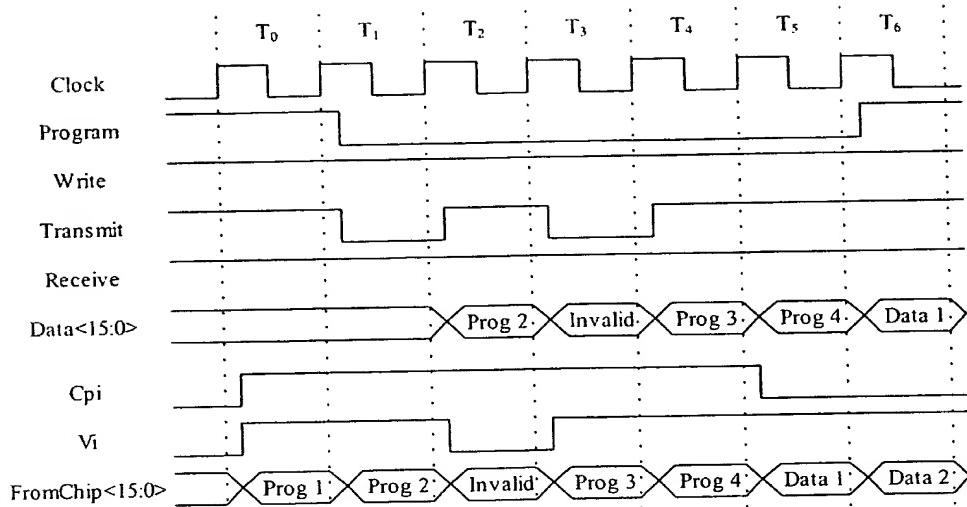
Figure A.12 - Programming a port as an output.

An example of programming the data port as an output from inside the chip is shown in

Figure A.12. The process begins during T_1 when the *ChipProgramIn (Cpi)* signal goes high. *Cpi*

is the program signal entering the data port from the crossbar, and *Vi* is the valid bit

(FromChip<16>) from the crossbar. The transition of *Cpi* to a logic 1 indicates the beginning of

new configuration information (a new stream header) and so the data port compares the address

of the first word in the stream to see if it is the correct address for the data port. If so, the first

word is latched by the data port as its own configuration word into the *Config<8:0>* register.

During T_1, the chip pulls the *Program* pin low, indicating the beginning of programming

information; however, *Write* remains high, thus the configuration information is coming out of

the data port and the Stream Controller must start accepting the new stream header. But, the first

word was stripped off of the header and used for the data port itself, thus the data port flags the

first word as being invalid by pulling the *Transmit* pin low. Meanwhile, the second configuration

301

word, *Prog 2*, is latched in the *Buffer<16:0>* of the data port. At time T_2, *Prog 2* is sent to the data pins by the port with the *Transmit* pin floating high. In this case, the Stream Controller is required to latch the word as valid configuration information. On the chip side of the data port, a word has arrived that has the *Vi* signal pulled low, indicating invalid data. The *ValidBitOut* signal will be low in this case and so a 0 will be latched into *Buffer<16>* to flag the invalid data. During T_3, the invalid data word reaches the pins, and the *Transmit* pin is pulled low to indicate this. At the same time, *Prog 3* is latched into the *Buffer<16:0>* and *Prog 4* is latched during T_4. At T_5, the stream header ends and the first data word is latched into *Buffer<16:0>*. This is indicated by *Cpi* being a logic 0. The *Program* pin reflects this change during T_6 as it is allowed to float high, indicating to the Stream Controller that the stream header has ended and the data section of the stream has begun. The *Write* pin floating high, indicates to the Stream Controller that a write is being performed. Further, the *Transmit* pin is high indicating that valid data is being written and and it should be latched. Operation proceeds using the same Transmit/Receive protocol from there.

These two cases typify the use of the data port. It is also possible to program a data port as an input from inside the chip, and it is possible to program a data port as an output from outside the chip. These were included mainly for completeness and are considered to be somewhat degenerate cases. They could be useful if it were possible to send a stream header to a data port from the crossbar and have the data port reflect the remainder of the stream back to the crossbar. However, there is the question of how the external Stream Controller would be programmed at that point, so this mode wasn't included, though perhaps it could be on future versions. Also, it is important to note that a configuration stream from outside the chip always

overrides a configuration stream that originates from inside the chip. Thus, if both of these scenarios occurred simultaneously, the stream originating from outside the chip would take precedence. It is important that the chip functions this way so that it is possible to always override any erroneous or reset condition within the chip from the outside.

A.4.3 State Machine

Below is the full logic description for the data port state machine. The interpretation of this follows.

```
 1    # Num. Vars, Num Bin Vars., Size Of Multi Value Vars
 2    .mv 15 14 12
 3
 4    # Names of binary variables
 5    .ilb   Q1   Q2   ProgramPinIn   WritePinIn   TransmitPinIn   ReceivePinIn   ChipProgramIn   ChipValidBitIn
 6    StartOfPacketMarker AddressCompare SynchronizationReadyIn RWBit SyncBit LoopBit
 7
 8    # Name parts of multi value var.
 9    .label var=14  Q1p Q2p  ProgramPinOut  WritePinOut  TransmitPinOut  ReceivePinOut  ChipProgramOut  ValidBitOut
10    RegBusDir PinBusDir SynchronizationReadyOut RWLoad
11
12    # Specifies that we will give ON-set, OFF-set and DC-set
13    .type fdr
14
15    # Current StateInput Vector   Output Vector
16    # Program 11
17    # Program Pin, My Sync Not Ready
18    11            000- -------- 11 1111 100000
19    # Program Pin, Running
20    11            001- -------- 11 1111 110000
21    # Chip Programming, My Sync Not Ready
22    11            -1-- 10------ 11 0101 001100
23    # Chip Programming, Running
24    11            -1-- 11------ 11 0111 001100
25    # Chip Programming, Somebody Pulled Write Pin Low, My Sync Not Ready
26    11            10-- 10------ 11 0101 001000
27    # Chip Programming, Somebody Pulled Write Pin Low, My Sync Ready
28    11            10-- 11------ 11 0111 001000
29
30    # These are the cases for doing a Read
31    # Programming Done, Non-Sync Mode, Read, My Sync Not Ready
32    11            1-0- 0----000 00 1011 000000
33    11            -10- 0----000 00 1011 000000
34    # Programming Done, Non-Sync Mode, Read, My Sync Ready
35    11            1-1- 0----000 00 1011 010010
36    11            -11- 0----000 00 1011 010010
37    # Programming Done, Sync Mode, Read, My Sync Not Ready
38    11            1-0- 0----010 01 1010 000000
39    11            -10- 0----010 01 1010 000000
40    # Programming Done, Sync Mode, Read, My Sync Ready, Chip Sync Not Ready
41    11            1-1- 0---0010 01 1010 000010
42    11            -11- 0---0010 01 1010 000010
43    # Programming Done, Sync Mode, Read, My Sync Ready, Chip Sync Ready
44    11            1-1- 0---1010 00 1011 010010
45    11            -11- 0---1010 00 1011 010010
46
47    # These are the cases for doing a Write in Non-Sync or Sync Mode
48    # Programming Done, Memory Not Ready, My Sync Not Ready, Write
49    11            11-0 00---1-0 00 1101 001100
50    11            01-0 00---1-0 00 1101 001100
51                  # Somebody Pulled Write Pin Low
52    11            10-0 00---1-0 00 1101 001000
```

303

```
53   # Programming Done, Memory Not Ready, My Sync Ready, Write
54   11          11-0 01---1-0 00 1111 001100
55   11          01-0 01---1-0 00 1111 001100
56               # Somebody Pulled Write Pin Low
57   11          10-0 01---1-0 00 1111 001000
58   # Programming Done, Memory Ready, My Sync Not Ready, Write
59   11          11-1 00---1-0 00 1101 001110
60   11          01-1 00---1-0 00 1101 001110
61               # Somebody Pulled Write Pin Low
62   11          10-1 00---1-0 00 1101 001010
63   # Programming Done, Memory Ready, My Sync Ready, Write
64   11          11-1 01---1-0 00 1111 001110
65   11          01-1 01---1-0 00 1111 001110
66               # Somebody Pulled Write Pin Low
67   11          10-1 01---1-0 00 1111 001010
68
69
70   # Programming Done, Loop Mode, Read, My Sync Not Ready
71   11          1-0- 0----0-1 11 1010 000000
72   11          -10- 0----0-1 11 1010 000000
73   # Programming Done, Loop Mode, Read, My Sync Ready, Chip Sync Not Ready
74   11          1-1- 0---00-1 10 1010 000010
75   11          -11- 0---00-1 10 1010 000010
76   # Programming Done, Loop Mode, Read, My Sync Ready, Chip Sync Ready
77   11          1-1- 0---10-1 10 1011 010010
78   11          -11- 0---10-1 10 1011 010010
79
80   # Programming Done, Loop Mode, Write, Memory Not Ready, My Sync Not Ready
81   11          11-0 00---1-1 11 1101 001100
82   11          01-0 00---1-1 11 1101 001100
83               # Somebody Pulled Write Pin Low
84   11          10-0 00---1-1 11 1101 001000
85   # Programming Done, Loop Mode, Write, Memory Not Ready, My Sync Ready
86   11          11-0 01---1-1 11 1111 001100
87   11          01-0 01---1-1 11 1111 001100
88               # Somebody Pulled Write Pin Low
89   11          10-0 01---1-1 11 1111 001000
90   # Programming Done, Loop Mode, Write, Memory Ready, Chip Sync Not Ready, My Sync Not Ready
91   11          11-1 00--01-1 11 1101 001110
92   11          01-1 00--01-1 11 1101 001110
93               # Somebody Pulled Write Pin Low
94   11          10-1 00--01-1 11 1101 001010
95   # Programming Done, Loop Mode, Write, Memory Ready, Chip Sync Not Ready, My Sync Ready
96   11          11-1 01--01-1 11 1111 001110
97   11          01-1 01--01-1 11 1111 001110
98               # Somebody Pulled Write Pin Low
99   11          10-1 01--01-1 11 1111 001010
100  # Programming Done, Loop Mode, Write, Memory Ready, Chip Sync Ready, My Sync Not Ready
101  11          11-1 00--11-1 00 1101 001110
102  11          01-1 00--11-1 00 1101 001110
103              # Somebody Pulled Write Pin Low
104  11          10-1 00--11-1 00 1101 001010
105  # Programming Done, Loop Mode, Write, Memory Ready, Chip Sync Ready, My Sync Ready
106  11          11-1 01--11-1 00 1111 001110
107  11          01-1 01--11-1 00 1111 001110
108              # Somebody Pulled Write Pin Low
109  11          10-1 01--11-1 00 1111 001010
110
111
112
113
114
115
116  # Active 00
117  #               These are the cases for programming that are common to all Non-Programming states
118  # Program Pin, Running, My Program Word
119  00          001- --11---- 11 1111 000001
120  00          011- 0-11---- 11 1111 000001
121  # Program Pin, My Sync Not Ready, Stay Here Until Get Valid Data Word
122  00          000- -------- 00 1111 000000
123  00          010- 0------- 00 1111 000000
124  # Program Pin, My Sync Ready, Skip Programming Port
125  00          001- --01---- 11 1111 110000
126  00          011- 0-01---- 11 1111 110000
127  00          001- --10---- 11 1111 110000
128  00          011- 0-10---- 11 1111 110000
129  00          001- --00---- 11 1111 110000
130  00          011- 0-00---- 11 1111 110000
131  # Chip Programming, Running, My Program Word
132  00          1--- 1111---- 11 0101 001001
133  00          01-- 1111---- 11 0101 001001
134  # Chip Programming, My Sync Not Ready, Stay Here Until Get Valid Data Word
```

304

```
135     00              1--- 10------ 00 0101 001000
136     00              01-- 10------ 00 0101 001000
137     # Chip Programming, My Sync Ready, Skip Programming Port
138     00              1 -- 1110---- 11 0111 001000
139     00              01-- 1110---- 11 0111 001000
140     00              1--- 1101---- 11 0111 001000
141     00              01-- 1101---- 11 0111 001000
142     00              1--- 1100---- 11 0111 001000
143     00              01-- 1100---- 11 0111 001000
144
145     #               These are the cases for doing a Read
146     # Read, Non-Sync Mode, My Sync Not Ready
147     00              1-0- 0----000 00 1011 000000
148     # Read, Non-Sync Mode, My Sync Ready
149     00              1-1- 0----000 00 1011 010010
150     # Read, Sync Mode, My Sync Not Ready
151     00              1-0- 0----010 01 1010 000000
152     # Read, Sync Mode, My Sync Ready, Chip Sync Not Ready
153     00              1-1- 0---0010 01 1010 000010
154     # Read, Sync Mode, My Sync Ready, Chip Sync Ready
155     00              1-1- 0---1010 00 1011 010010
156
157
158     #               These are the cases for doing a Write
159     # Write, But Somebody Pulled Write Pin Low, Memory Not Ready, My Sync Not Ready
160     00              10-0 00---1-0 00 1101 001000
161     # Write, But Somebody Pulled Write Pin Low, Memory Not Ready, My Sync Ready
162     00              10-0 01---1-0 00 1111 001000
163     # Write, But Somebody Pulled Write Pin Low, Memory Ready, My Sync Not Ready
164     00              10-1 00---1-0 00 1101 001010
165     # Write, But Somebody Pulled Write Pin Low, Memory Ready, My Sync Ready
166     00              10-1 01---1-0 00 1111 001010
167
168     # Write, Memory Not Ready, My Sync Not Ready
169     00              11-0 00---1-0 00 1101 001100
170     # Write, Memory Not Ready, My Sync Ready
171     00              11-0 01---1-0 00 1111 001100
172     # Write, Memory Ready, My Sync Not Ready
173     00              11-1 00---1-0 00 1101 001110
174     # Write, Memory Ready, My Sync Ready
175     00              11-1 01---1-0 00 1111 001110
176
177
178     # Loop Mode Stuff
179     # Loop Mode, Read, Can't Happen
180     00              1--- 0----0-1 -- ---- ---0--
181
182     # Loop Mode, Write, But Somebody Pulled Write Pin Low, My Sync Not Ready
183     00              10-- 00---1-1 00 1101 001000
184     # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Not Ready, My Sync Ready
185     00              10-0 01---1-1 10 1111 001000
186     # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, My Sync Ready, Chip Sync Not Ready
187     00              10-1 01--01-1 10 1111 001010
188     # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, My Sync Ready, Chip Sync Ready
189     00              10-1 01--11-1 00 1111 001010
190
191
192     # Loop Mode, Write, My Sync Not Ready
193     00              11-- 00---1-1 00 1101 001100
194     # Loop Mode, Write, Memory Not Ready, My Sync Ready
195     00              11-0 01---1-1 10 1111 001100
196     # Loop Mode, Write, Memory Ready, My Sync Ready, Chip Sync Not Ready
197     00              11-1 01--01-1 10 1111 001110
198     # Loop Mode, Write, Memory Ready, My Sync Ready, Chip Sync Ready
199     00              11-1 01--11-1 00 1111 001110
200
201
202
203
204     # Deassert Sync 01
205     #               These are the cases for programming that are common to all Non-Programming states
206     # Program Pin, Running, My Program Word
207     01              001- --11---- 11 1111 000001
208     01              011- 0-11---- 11 1111 000001
209     # Program Pin, My Sync Not Ready, Stay Here Until Get Valid Data Word
210     01              000- -------- 01 1111 000000
211     01              010- 0------- 01 1111 000000
212     # Program Pin, My Sync Ready, Skip Programming Port
213     01              001- --01---- 11 1111 110000
214     01              011- 0-01---- 11 1111 110000
215     01              001- --10---- 11 1111 110000
216     01              011- 0-10---- 11 1111 110000
```

305

```
217    01           001- --00---- 11 1111 110000
218    01           011- 0-00---- 11 1111 110000
219    # Chip Programming, Running, My Program Word
220    01           1--- 1111---- 11 0101 001001
221    01           01-- 1111---- 11 0101 001001
222    # Chip Programming, My Sync Not Ready, Stay Here Until Get Valid Data Word
223    01           1--- 10------ 01 0101 001000
224    01           01-- 10------ 01 0101 001000
225    # Chip Programming, My Sync Ready, Skip Programming Port
226    01           1--- 1110---- 11 0111 001000
227    01           01-- 1110---- 11 0111 001000
228    01           1--- 1101---- 11 0111 001000
229    01           01-- 1101---- 11 0111 001000
230    01           1--- 1100---- 11 0111 001000
231    01           01-- 1100---- 11 0111 001000
232
233    # Write - Can't Happen
234    01           1--- 0----1-- -- ---- ------
235    # Read, My Sync Not Ready
236    01           1-0- 0----0-- 01 1010 000000
237    # Read, My Sync Ready, Chip Sync Not Ready
238    01           1-1- 0---00-- 01 1010 000010
239    # Read, My Sync Ready, Chip Sync Ready
240    01           1-1- 0---10-- 00 1011 010010
241
242
243
244
245    # Deassert Loop 10
246
247    #            These are the cases for programming that are common to all Non-Programming states
248    # Program Pin, Running, My Program Word
249    10           001- --11---- 11 1111 000001
250    10           011- 0-11---- 11 1111 000001
251    # Program Pin, My Sync Not Ready, Stay Here Until Get Valid Data Word
252    10           000- -------- 10 1111 000000
253    10           010- 0------- 10 1111 000000
254    # Program Pin, My Sync Ready, Skip Programming Port
255    10           001- --01---- 11 1111 110000
256    10           011- 0-01---- 11 1111 110000
257    10           001- --10---- 11 1111 110000
258    10           011- 0-10---- 11 1111 110000
259    10           001- --00---- 11 1111 110000
260    10           011- 0-00---- 11 1111 110000
261    # Chip Programming, Running, My Program Word
262    10           1--- 1111---- 11 0101 001001
263    10           01-- 1111---- 11 0101 001001
264    # Chip Programming, My Sync Not Ready, Stay Here Until Get Valid Data Word
265    10           1--- 10------ 10 0101 001000
266    10           01-- 10------ 10 0101 001000
267    # Chip Programming, My Sync Ready, Skip Programming Port
268    10           1--- 1110---- 11 0111 001000
269    10           01-- 1110---- 11 0111 001000
270    10           1--- 1101---- 11 0111 001000
271    10           01-- 1101---- 11 0111 001000
272    10           1--- 1100---- 11 0111 001000
273    10           01-- 1100---- 11 0111 001000
274
275    # Read, My Sync Not Ready
276    10           1-0- 0----0-1 10 1010 000000
277    # Read, My Sync Ready, Chip Sync Not Ready
278    10           1-1- 0---00-1 10 1010 000010
279    # Read, My Sync Ready, Chip Sync Ready
280    10           1-1- 0---10-1 10 1011 010010
281
282
283    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Not Ready, My Sync Not Ready
284    10           10-0 00---1-1 10 1101 001000
285    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Not Ready, My Sync Ready
286    10           10-0 01---1-1 10 1111 001000
287    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, Chip Sync Not Ready, My Sync Not Ready
288    10           10-1 00---01-1 10 1101 001010
289    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, Chip Sync Not Ready, My Sync Ready
290    10           10-1 01---01-1 10 1111 001010
291    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, Chip Sync Ready, My Sync Not Ready
292    10           10-1 00--11-1 10 1101 001010
293    # Loop Mode, Write, But Somebody Pulled Write Pin Low, Memory Ready, Chip Sync Ready, My Sync Ready
294    10           10-1 01--11-1 00 1111 001010
295
296
297    # Loop Mode, Write, Memory Not Ready, My Sync Not Ready
298    10           11-0 00---1-1 10 1101 001100
```

306

```
299    # Loop Mode, Write, Memory Not Ready, My Sync Ready
300    10           11-0 01---1-1 10 1111 001100
301    # Loop Mode, Write, Memory Ready, Chip Sync Not Ready, My Sync Not Ready
302    10           11-1 00 -01-1 10 1101 001110
303    # Loop Mode, Write, Memory Ready, Chip Sync Not Ready, My Sync Ready
304    10           11-1 01--01 1 10 1111 001110
305    # Loop Mode, Write, Memory Ready, Chip Sync Ready, My Sync Not Ready
306    10           11-1 00--11-1 00 1101 001110
307    # Loop Mode, Write, Memory Ready, Chip Sync Ready, My Sync Ready
308    10           11-1 01--11-1 00 1111 001110
309
310
311    # Non-Loop Mode, Can't Happen
312    10           ---- -------0 -- ---- ------
```

Figure A.13 – Data Port State Machine Transitions.

The control of the data port centers around the state machine. Though it has only four possible states, there are 154 unique transitions between those states, thus it was not tractable to draw a state transition diagram. Nor was it feasible to do the design entirely by hand. The *misII* [83] logic minimization program was used produce the output equations required. This was done by separating the purely combination part of the state machine from the flip flops used to store the state, and then putting the truth table for the excitation equations into a format that misII can read (a PLA file). As far as misII is concerned, the state machine is then a multiple input multiple output logic minimization problem.

Because of the large number of separate transitions involved, it was difficult to determine when transition minterms overlapped and when some input combinations were not defined. To combat this problem, the author wrote a program called *chkcover* that performs the first stage of the Quine-McCluskey logic minimization algorithm in order detect both error conditions. *Chkcover* accepts the PLA file as input, analyzes covered terms and reports overlap and unspecified combinations. The input file containing the logic function descriptions is shown in Figure A.13.

Table A.3 - Data Port State Machine Inputs.

Abrv.	Name	Sense
Pp	Program Pin	AL
W	Write Pin	AH
Tx	Transmit Pin	AH
Rx	Receive Pin	AH
Cpi	Chip Program In	AH
Vi	Chip Valid Bit In	AH
M	Start Of Packet Marker	AH
A	Address Compare	AH
Si	Synchronization Ready In	AH
Rwb	Read/Write Bit	AH
Sb	Sync Bit	AH
Lb	Loop Bit	AH

Table A.4 - Data Port State Machine Outputs.

Abrv.	Name	Sense
Pp	Program Pin	AL
W	Write Pin	AH
Tx	Transmit Pin	AH
Rx	Receive Pin	AH
Cpo	Chip Program Out	AH
Vo	Valid Bit Out	AH
RDir	RegBusDir	Out=1
PDir	PinBusDir	Out=1
So	Synchronization Ready Out	AH
Rwl	Read/Write Flag Load	AH

The .ilb line near the top of the PLA file specifies the names of the 14 inputs, including

Q1 and *Q2* (the current state). These are also listed in Table A.3, along with their abbreviations

---

83 Multiple-Level Combinational Logic Optimization Program, Miscellaneous Reference

and sense of use: AH for active high and AL for active low. Table A.4 and the .label line in the PLA file specify the names of the 12 outputs, including $Q1p$ and $Q2p$ (the next state). Finally, Table A.5 shows the state assignment used for the data port state machine.

Table A.5 - Data Port State Machine State Assignment.

State	Assignment
Program	11
Active	00
Deassert Sync	01
Deassert Loop	10

The PLA file in Figure A.13 can be read as follows. The current state is indicated by the two bits in the far left column. Next are the inputs to the state machine, with the four flow control pins separated in one field and all the other miscellaneous pins listed in the column to their right. The next state is listed in the fourth column and the fifth column shows the values that the state machine will drive the four control pins to during the next clock period. Lastly, the sixth column shows the values that the state machine will produce for the output signals. Note that the state machine is a kind of Mealy and Moore machine hybrid. As the outputs are produced from the PLA file, the state machine should be considered to be a Mealy machine. However, all the combinational logic shown in the PLA file is contained entirely within the red box in Figure A.10. Note that some of these signals are latched before they actually leave the confines of the data port. Thus, the *ProgramPinOut*, *WritePinOut*, *TransmitPinOut*, *ReceivePinOut*, *ChipProgramOut* and *ValidBitOut* signals are all latched before entering the chip

as a whole. Further, some of these signals form feedback paths to the state machine and the time delay introduced by the flip flops must be considered in the construction of the PLA file.

There are actually three separate state machines superimposed on one another in the PLA file; one for each of the three modes of operation of the data port. The comments, indicated by lines starting with the # character, give some hint as to which mode of operation each line is intended to aid. Lines with comments containing Loop Mode are for Loop Mode. Lines whose comments mention Sync Mode are for Synchronization Mode and lines that indicate Non-Sync mode are for Raw Mode. Lines that have comments dealing with programming are common to all three modes of operation. Chip Programming indicates that the line is for a stream header originating from inside the chip and Pin Programming indicates that the stream header originated from outside the chip. Also, when My Sync is mentioned in the comments, it is referring to the "valid bit" input at that point. Thus, if the stream is originating from off chip, the "valid bit" input is the *Transmit* pin, since that is what is used to flag invalid data. Likewise, when the stream is originating from inside the chip the "valid bit" input is the actual valid bit coming from the data on the crossbar, *FromChip<16>* or *Vi*. The comments also mention Chip Sync, which is used for both Synchronization and Loop Modes. It is actually the *SynchronizationReadyIn* signal, which is generated from the masking operation of the *SynchronizationReadyOut* signals from the other data ports. All the *SynchronizationReadyOut* signals are bundled into the *SyncBus<5:0>*. Finally, perhaps obviously, the words Read or Write indicate the programmed direction of data flow through the port.

```
Q1p = ChipValidBitIn * LoopBit * RegBusDir * !SynchronizationReadyOut +
      !SynchronizationReadyIn * LoopBit * RegBusDir * SynchronizationReadyOut
      + Q1 * LoopBit * !SynchronizationReadyOut + TransmitPinIn * WritePinOut
      * !RegBusDir + Q1 * WritePinOut * !RegBusDir + Q1 * LoopBit * !RegBusDir
      + ChipProgramIn * ChipValidBitIn * RegBusDir + Q1 * ChipProgramIn;
Q2p = Q2 * Q1p * !SynchronizationReadyOut + TransmitPinIn * WritePinOut *
      !RegBusDir + ChipValidBitIn * !ProgramPinOut * RegBusDir + Q2 * Q1p *
      RegBusDir + !Q1p * ProgramPinOut * !ReceivePinOut + !Q1 * Q2 *
      WritePinOut;
ProgramPinOut = !Q1 * !RWBit * !LoopBit * !RegBusDir + Q1 * LoopBit *
      !RegBusDir + Q2 * !RWBit * !RegBusDir + WritePinOut * !RegBusDir +
      !ChipProgramIn * RegBusDir;
WritePinOut = Q1 * RWBit * LoopBit + !Q1 * !Q2 * RWBit + Q1 * Q2 * RWBit +
      ProgramPinIn * !WritePinIn + !Q2 * !ProgramPinIn + !Q1 * !ProgramPinIn +
      ChipProgramIn;
TransmitPinOut = ChipValidBitIn * !AddressCompare * RegBusDir + ChipValidBitIn
      * !StartOfPacketMarker * RegBusDir + Q1 * Q2 * ChipValidBitIn +
      ProgramPinOut * !RegBusDir + ChipValidBitIn * ProgramPinOut;
ReceivePinOut = !Q2 * !SyncBit * !LoopBit * ProgramPinOut + Q1 * !SyncBit *
      !LoopBit * ProgramPinOut + TransmitPinIn * SynchronizationReadyIn *
      ProgramPinOut + WritePinOut;
ChipProgramOut = !Q2 * !ProgramPinIn * TransmitPinIn * !ChipProgramIn *
      !AddressCompare + !Q1 * !ProgramPinIn * TransmitPinIn * !ChipProgramIn *
      !AddressCompare + !Q2 * !ProgramPinIn * TransmitPinIn * !ChipProgramIn *
      !StartOfPacketMarker + !Q1 * !ProgramPinIn * TransmitPinIn *
      !ChipProgramIn* !StartOfPacketMarker + !ProgramPinIn * !WritePinIn *
      TransmitPinIn * !AddressCompare + !ProgramPinIn * !WritePinIn *
      TransmitPinIn * !StartOfPacketMarker + Q1 * Q2 * !ProgramPinIn*
      !WritePinIn;
ValidBitOut = TransmitPinIn * !WritePinOut * ReceivePinOut + TransmitPinIn *
      ChipProgramOut;
RegBusDir = ProgramPinIn * WritePinOut + WritePinIn * ChipProgramIn +
      PinBusDir;
PinBusDir = ProgramPinIn * WritePinIn * !ChipProgramIn * WritePinOut + Q1 * Q2
      * WritePinIn * WritePinOut;
SynchronizationReadyOut = ReceivePinIn * !LoopBit * ProgramPinOut * RegBusDir
      + ReceivePinIn * ChipValidBitIn * ProgramPinOut * RegBusDir + Q1 *
      ReceivePinIn * ProgramPinOut * RegBusDir + TransmitPinIn * ProgramPinOut
      * !WritePinOut;
RWLoad = !Q2 * WritePinIn * ChipProgramIn * ChipValidBitIn *
      StartOfPacketMarker * AddressCompare + !Q1 * WritePinIn * ChipProgramIn
      * ChipValidBitIn * StartOfPacketMarker * AddressCompare + !Q2 *
      ProgramPinIn * ChipProgramIn * ChipValidBitIn * StartOfPacketMarker *
      AddressCompare + !Q1 * ProgramPinIn * ChipProgramIn * ChipValidBitIn *
      StartOfPacketMarker * AddressCompare + !Q2 * !ProgramPinIn *
      TransmitPinIn * !ChipProgramIn * StartOfPacketMarker * AddressCompare +
      !Q1 * !ProgramPinIn * TransmitPinIn * !ChipProgramIn *
      StartOfPacketMarker * AddressCompare + !Q2 * !ProgramPinIn * !WritePinIn
      * TransmitPinIn * StartOfPacketMarker * AddressCompare + !Q1 *
      !ProgramPinIn * !WritePinIn * TransmitPinIn * StartOfPacketMarker *
      AddressCompare;
```

Figure A.14 - Data Port State Machine Equations.

A.4.3.1 Description of Signals

A complete description of all the state machine inputs and outputs seems appropriate. As listed in Table A.3, the inputs and their functions are:

ProgramPinIn - This is the current value of the *Program* pin for the data port. The pin is implemented as an open drain circuit, thus either the chip or an external force can pull the pin to a logic 0 at any time. A value of 0 on this pin indicates that some type of programming information is passing through the data port. A value of 1 indicates that data is being transferred through the port.

WritePinIn - This is the current value of the *Write* pin for the data port. This pin is implemented in the same open drain fashion as the *Program* pin. A value of 0 on the *Write* pin indicates that the data on the 16-bit data bus is flowing from the pins to the inside of the chip. A value of 1 indicates that data is moving from the inside of the chip to the pins.

TransmitPinIn - The *Transmit* pin is also implemented as an open drain pin, and this is its current value. A logic 0 on this pin indicates that the value being sent from the data source is invalid and should be ignored. A logic 1 indicates that the data being transferred is good and should be processed normally.

ReceivePinIn - This is the current value of the last of the four flow control pins, *Receive*. Again, *Receive* is implemented as an open drain circuit. A logic 0 on this pin indicates that the data sink is either full or nearly full and every attempt should be made to stop new data from being transferred, otherwise it may be dropped. A logic 1 means that data transfer may proceed as normal.

ChipProgramIn - This is the program signal associated with the data values coming to the data port from the crossbar. When this signal is a logic 1 it indicates that the associated data word should be interpreted as configuration data. A logic 0 implies that the data word from the crossbar is normal operand data.

ChipValidBitIn - This is the valid bit associated with the data word coming to the data port from the crossbar. It is normally bit 16 of any data word while it is internal to the chip; however, it was pulled out separately for signal labeling purposes. If this bit is a logic 0, it means that the associated data word is junk and should be ignored. If the bit is a logic 1, then the data is good and should be processed normally.

StartOfPacketMarker - This bit is used to indicate the first word of a programming packet. It is actually bit 15 of every configuration word. If the bit is a 1, then this is the first word of a configuration packet. If the bit is a 0, this is not the first word of a programming packet.

AddressCompare - Is the result of applying the lower six bits of the word currently being latched by the *Buffer<16:0>* to the address comparator circuit for the unit. If the lower six bits either match the address for the data port, or match the broadcast address (111111), the value will be a 1, otherwise it will be a 0.

SynchronizationReadyIn - This signal is used to determine when the flow control signals for a programmer defined set of data ports will allow processing to continue. The *SyncBus<5:0>* carries the *SynchronizationReadyOut* signals from all six of the data ports to every data port. Each of these is NORed with a bit from a programmer specified mask, stored in the data port configuration register. If the mask bit is a 1, then that synchronization signal is ignored. If the

313

mask bit is a 0, the bit is included in the comparison. If all the synchronization bits included by the mask have a logic 1 value (all the data ports in the set are "ready"), then the value of *SynchronizationReadyIn* will be a 1. Otherwise, it will be a 0.

Read/WriteBit (RWBit) - This is the value of the configuration bit that controls the direction of the port during operand data processing. If the bit is a logic 1, then the port is to function in write mode. If the bit is a logic 0 the port is to function in read mode.

SyncBit and LoopBit - These bits together determine what mode of operation the data port will function in during operand processing. If both bits are 0, then the data port will function in Raw Mode. If LoopBit is a 1, then the data port will function in Loop Mode. If *LoopBit* = 0 and *SyncBit* = 1 the port will function in Synchronization Mode. Both of these bits are also stored in the data port configuration register.

The outputs from the data port state machine, as listed in Table A.4, and their functions are:

ProgramPinOut, WritePinOut, TransmitPinOut, ReceivePinOut - These are the control signals from the data port to the indicated open drain pins. If the value of one of these signals is a logic 0, the corresponding pin will attain that value during the following clock pulse. The pin is delayed one cycle from the value issued from the state machine because the signal is latched before it reaches the pin. If the state machine sets this pin to a logic 1, then it will attain that value if no other sources are pulling the pin to a logic 0, since it is open drain. The meaning of each pin was described above.

ChipProgramOut - This is the program signal associated with the data value going from the data port to the crossbar. If it is a logic 1, then the data value is configuration information. If the signal is a logic 0, the data value is normal data.

ValidBitOut - This is the value to be latched by the *Buffer<16:0>* register into bit 16 for use as the data value's valid bit. If the valid bit is a 0, then the data value will be ignored. If the valid bit is a 1, the data value will be processed normally.

RegBusDir - This is the direction of data flow through the registers of the data port. If the signal is a logic 0, then data port is functioning in input mode and data from the pins is latched into the *Buffer<16:0>* register. If the signal is a logic 1, the data port is functioning in output mode and data from the crossbar is latched into the *Buffer<16:0>* register.

PinBusDir - This is the orientation of the 16 data pins themselves. If the signal is a logic 0, then the pins are tri-stated so that they can safely function as inputs. If the signal is a logic 1, the pins actively drive the signal coming from the *Buffer<16:0>* register.

SynchronizationReadyOut - This is used to signal when this data port is ready to continue processing according to the current state of the flow control pins and the direction of data flow. Exactly what this means depends on the data port mode and the programmed direction. The signal is sent to all the other data ports so that any data port may synchronize with any other. Upon arrival at a given data port, the signal is masked and combined with the *SynchronizationReadyOut* signals of other data ports to form the *SynchronizationReadyIn* signal discussed above. If the *SynchronizationReadyOut* signal is a logic 1, it indicates that this data port is "ready." The port is not ready otherwise.

315

Read/Write Flag Load (RWLoad) - This signal controls the load signal for the register that stores the configuration information for the data port. If this signal is a logic 1, the register will latch the data that is currently being presented to the inputs of the *Buffer<16:0>* register. If the signal is a logic 0, no latching will take place.

A.4.3.2 Programming

There are essentially two main sections in the PLA file concerned with programming. Upon reset the state machine enters the *Active* state (00) in a Raw Mode read configuration and will remain there until programmed. During normal port operation, the state machine will transition between the *Active* (00), *Deassert Sync* (01) and *Deassert Loop* (10) states. Programming may begin at any time, and thus, the same programming prelude lines are included as transitions from all three of these states to the 11 state. The three copies are shown in lines 117-143, 205-231 and 247-273. The first state shown in the file, 11 (lines 16-109), is used after programming has begun. While in that state, the data port will continue to accept programming information and will forward it either into or out of the chip.

As an example of programming behavior, line 119 is reproduced here:

00   001- --11---- 11 1111 000001

The first two bits are 00, indicating that it is from the *Active* state and so is one of the entry points to the *Programming* state (11). The "trigger" conditions on the data port pins for this transition are that *Program* and *Write* must be low and *Transmit* must be high, indicating that an external stream header is being injected into the chip and that the word present on the pins is valid. The *Receive* pin is a don't care for this transition because in this case it is the signal from the data port

316

to the Stream Controller to stop sending data. A stream header is always accepted with no attempt at flow control. The actual state of the *Receive* pin is in fact unknown at this point because the previous operational state of the data port is unknown. *ChipProgramIn* and *ChipValidIn* are the next signals and they are both don't cares because external streams always take precedence over internal streams. Both *StartOfPacketMarker* and *AddressCompare* are required to be at logic 1 for this transition to be used. *StartOfPacketMarker* is bit 15 of the incoming configuration word, and as mentioned above, when 1 it signals the first word in a new packet. *AddressCompare* is the result of the address comparator for the address contained in this configuration word. If *AddressCompare* is a 1, it indicates that the address matches.

Under these conditions, it is known that a valid configuration word is being injected from outside of the chip and that it is the first word of a new packet, whose address matches that of the data port. This being the case, the data port will latch the word into its own configuration storage register *Config<8:0>*, and transition to the *Programming* state (11). The outputs reflect this action. First, the next state transition is 11. Also, the *Program, Write* and *Transmit* port pins are given up to external control by setting them to a logic 1. The *Receive* pin should technically be under the control of the data port since it is the reverse going flow control signal in this case, but a stream header is always accepted unconditionally, so it is set to a logic 1 value. The *ChipProgramOut* signal is left at a logic 0 for the next clock pulse because it will be associated with this programming word, which was intended for the data port, thus there is no reason to forward it to the rest of the chip as such. Further, the *ValidBitOut* signal is set to a logic 0 as well, because although the *Cpo* signal is logic 0 and so the word would be considered to be data, it will be sent to the crossbar and so should be considered to be invalid so that it doesn't interfere

317

with other operations. Both *RegBusDir* and *PinBusDir* are set to logic 0 so that the direction of data flow through the data port registers and the direction of the data pins, respectively, are from the outside of the chip to the inside. *SynchronizationReadyOut* is a logic 0 because no data synchronization is being performed. Finally, the Read/Write Flag Load (*RWLoad*) signal is set to a logic 1 so that the *Config<8:0>* register will load the new configuration word.

A.4.3.3 Raw Mode

Raw Mode is the simplest of the data port operational modes. In Raw Mode, no attempt is made to provide flow control and data is merely accepted and passed on by the port. Thus, if configured as an input, *ReceivePinOut* is always held high by the port. If the last value written was valid (*TransmitPinIn* = 1), the data will be passed to the rest of the chip with the valid bit set to 1, as shown in line 149. Naturally, if the *TransmitPinIn* signal is low the valid bit will be set to 0, as shown in line 147. Also, the *SynchronizationReadyOut* signal is used as would be expected so that the port can be polled for use in synchronization even though the port itself does not use flow control.

The lines used for a port configured to write in Raw Mode are found on lines 168-175. These same lines are also used for the same function in Synchronization Mode. This is possible because an output port cannot buffer data if the Stream Controller connected to that port signals that data cannot be accepted. The port has no choice but to write out any valid data that it receives from the chip and so if valid data arrives it is sent to the pins with the *TransmitPinOut* signal set to 1. An output port can; however, toggle the *SynchronizationReadyOut* signal to a logic 0 in the event that the *ReceivePinIn* goes low in the hope that the input ports will stop

318

allowing new data to enter the chip. This is precisely what happens in Synchronization and Loop Modes.

Lines 158-166 are included for the odd ball case in which a data port has been programmed as an output, but the *WritePinIn* signal has been pulled low externally. To prevent burnout of the data pin drivers due to contention, these transitions all immediately reverse the direction of the *PinBus*, by setting *PinBusDir* to a logic 0. In all other ways, the transitions do mimic the same behavior as found in lines 168-175 in case the situation was merely caused by a glitch. Note that the only instance in which the *WritePin* should be pulled low by an external source is in the event that the Stream Controller wishes to initiate a new stream (configuration information) from the outside of the chip, and in that case *ProgramPin* should also be pulled low. Since *ProgramPin* is high for these transitions the condition is considered to be controller error.

A.4.3.4 Synchronization Mode

Synchronization Mode is used for standard stream flow control. It allows all data ports in a specified set to be configured to cooperate; only allowing data to enter the chip when all input and output ports are receiving "My Sync" (or "Ready") signals from their associated Stream Controllers. For the input ports, this means that all the *TransmitPinIn* signals are at a logic 1 value. If any of the *TransmitPinIn* signals for the input data ports in the set are pulled to a logic 0, then all ports specified will pull their *ReceivePinOut* signals low during the following clock period, indicating that none of them accepted the last data value and that it should be resent indefinitely. Likewise, if the *ReceivePinIn* signals for any of the output data ports in the set are

319

pulled low, it will cause all input ports to stop accepting data so that only the operands already in the on-chip pipeline will then be written out from the outputs. As discussed previously, the output data ports must write out any valid data item arriving at them from the chip and the on-chip pipeline cannot be stopped; thus, stopping the input ports from allowing data to enter is the best form of flow control available from that point.

The set of data ports that are synchronized in this mode, and in Loop Mode, are specified using a mask that is given as part of the configuration data. The six bit mask contains a bit position for each of the six data ports on the chip. If a bit in this mask is 0 for a given data port, then the *SynchronizationReadyOut* signal from that data port is used to calculate the *SynchronizationReadyIn* signal for this port and so it will play a part in the flow control process. If a given data port is to be ignored, then its bit position in the mask should be set to a logic 1. If the *SynchronizationReadyOut* signals for all data ports whose mask bits are set to 0 are all 1's, then the *SynchronizationReadyIn* signal for the port will be a 1, and the state machine can proceed with normal processing. If the *SynchronizationReadyIn* signal is a logic 0 and the data port is in input mode, then the data port should stop accepting data from the Stream Controller until *SynchronizationReadyIn* goes back to a logic 1 value.

The flow of state transitions for Synchronization Mode proceeds as follows for a port in output mode. At the end of programming, the state machine is in the *Programming* (11) state. The state machine will then transition to the *Active* (00) state, where it will stay until reconfigured again. While in the *Active* state, the state of the *ReceivePinIn* signal will be duplicated on the *SynchronizationReadyOut* signal. This will allow any input ports that are involved in the same computation to monitor the signals sent from that Stream Controller so that

320

the flow of data into the chip may be stopped, if necessary. As mentioned in the previous section, the same transitions (lines 145-166) are used for both Raw Mode and for Synchronization Mode, since no special behavior is required on the part of the output ports.

The behavior of a data port that is configured as an input for Synchronization Mode is somewhat more complicated. Under ideal conditions, the state machine for such a port will also transition from the *Program* state (11) to the *Active* state (00) and remain there until reconfigured. However, if any of the ports specified in the mask for that port loose their "My Sync" signals from the stream controllers, the *SynchronizationReadyIn* signal for the port will go to 0 and the data port must attempt to stop data from entering the chip. If either the *TransmitPinIn* or the *SynchronizationReadyIn* signal for an input data port go to 0, the port will go to the *Desassert Sync* state (01) until all data ports in the specified set have regained the "My Sync" signal and both the *TransmitPinIn* ("My Sync") and *SynchronizationReadyIn* ("Chip Sync") signals have gone back to a logic 1 value. Once conditions have returned to normal, all the data port state machines will transition back to the *Active* state (00) and will begin accepting data again.

A.4.3.5 Loop Mode

Loop Mode guarantees that only one set of valid operands exists within the chip pipeline at any given time. This guarantee allows the construction of pipelines that loop back on themselves within the chip so that data dependent looping can be performed. If more than one set of valid operands were admitted to such a pipeline, collisions and misalignments could occur because the number of loop traversals from one data item to the next may differ. Such looping

321

could be used for the calculation of a factorial for example, or for a successive approximation calculation. In practical terms, a set of input ports that are configured in Loop Mode will each simultaneously allow a single valid operand to enter the chip. None will accept another valid operand until a valid result leaves through a specified output port.

The state transitions used for Loop Mode are similar to those used for Synchronization Mode. At the end of configuration, all input ports wait in the *Programming* state (11) until all ports specified in the set have the "My Sync" signal from their Stream Controllers. At that point, all input ports will admit a single valid operand and will then enter the *Deassert Loop* state (10). These will wait here until their *TransmitPinIn* ("My Sync") and *SynchronizationReadyIn* ("Chip Sync") signals are both logic 1, at which point each will admit another valid data item. They will admit a valid data item for every clock cycle in which both conditions are true. The output data ports guarantee that the *SynchronizationReadyIn* signal will only be a logic 1 for a single clock period because they only raise their *SynchronizationReadyOut* signals when valid data arrives at the port from the chip. Since only one operand was admitted, the data ports should only see a single valid data item departing and will then assert *SynchronizationReadyOut* for a single clock period.

The output data ports also wait at the end of configuration in the *Programming* (11) state until all ports specified in the configuration mask have received the "My Sync" signal. They will then enter the *Active* state (00) and stay there with *SynchronizationReadyOut* set to 0 until receiving a valid operand from the chip. When an output port receives a valid operand from the chip, the state machine asserts *SynchronizationReadyOut* and then transitions to the *Deassert Loop* state (10). An output port in the Deassert Loop state continues to hold

322

SynchronizationReadyOut high as a flag indicating that a valid operand has been written from the chip. When all output data ports specified in the configuration mask have written a valid operand, the *SynchronizationReadyIn* signal for these, and the input ports, will go to logic 1, a single new operand will be admitted at each of the input ports and all of the output ports will return to the *Active* state (00), lowering their *SynchronizationReadyOut* signals; causing *SynchronizationReadyIn* to lower for all ports involved.

A.4.3.6 Programming Peculiarities

There are a number of special conditions that must be considered when programming the Colt. The first of these is initial stream synchronization. When programming two intersecting streams, it is possible to completely program one stream before programming the second and have the first begin operation, processing valid operands that do not originate from the second stream. To prevent this, the Stream Controllers must delay the data sections of all input streams for a given chunk until all streams have been at least partially programmed. This is accomplished by holding the *TransmitPin* of each port involved low until the appointed time. A solution to this problem is given in the future directions section of the conclusions, but it was not implemented on the Colt.

A related problem of programming initialization can crop up with Loop Mode. In Loop Mode, all ports wait in the *Programming* state until all have gotten the "My Sync" signal back from the Stream Controllers (indicated by *SynchronizationReadyIn* going high). Then a single valid operand is admitted through each of the input ports. It is possible to send a stream header into the Colt that is very short so that the input port will finish programming and enter the *Active*

323

state before all the output ports have been programmed. In this case, the output ports may assert *SynchronizationReadyOut*; inadvertently signaling the input ports to admit data and destroying the synchronization mechanism. This can be prevented by padding the end of the stream header (programming data) with a series of NOPs that will guarantee that the output ports are programmed before the input ports reach the end of the stream header. A NOP has the *StartOfPacketMarker* bit set (bit 15) and has an address of 000000 (bits 5-0). Alternatively, the same initialization sequence used by the Stream Controllers to solve the problem mentioned in the previous paragraph could also be tied into the *ReceivePin* of each of the output data ports, making sure that each is held low until all paths have been programmed.

A true peculiarity of operation that should never arise under real conditions occurs when programming a data port from the chip side of the port as an input. If this is done and the last word of the stream header is the one used to configure the port, it will be written to the data pins as an invalid programming word, as would normally occur if the port were being programmed as an output. However, since the port is programmed it should be possible to use the port as an input the first cycle after configuration, but because of this single invalid word it is not possible. After this first cycle, the port can be used as an input as normal. This is a degenerate case since it violates the stream concept by not configuring the device in the direction of data flow.

A.4.4 Electrical Design

The four bi-directional control pins: *Program*, *Write*, *Receive* and *Transmit*, were the subject of some concern because they had to be implemented as open collector drivers in order to

function properly. There was some question as to whether or not the pull up resistor could drive the bus high in the space of the 20ns clock period given by a chip running at 50MHz.
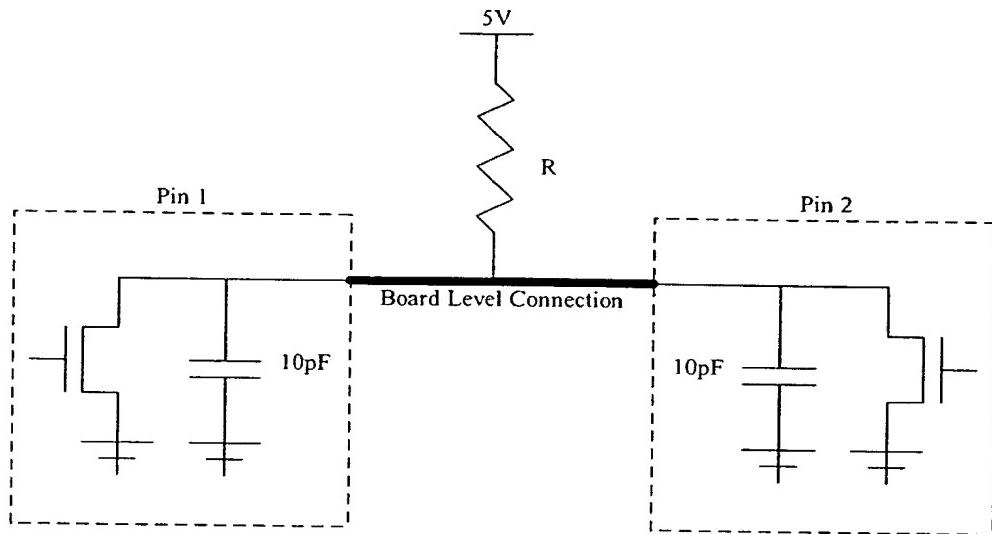
Figure A.15 – Bi-Directional Pin Model.

The circuit in question can be modeled as in Figure A.15. The 10pF values for the pin capacitance were taken from values given for several chips in data books. The first question is whether or not the resistor can drive the wire back up to 5V from 0V if both of the nFETs release control. This is governed by the normal capacitance equation $V = Vi(1 - e^{-t/(RC)})$. In this case, we are interested in what size resistor will allow the wire to be pulled back up to 2.5V, (the switching point of the pin drivers) in 10ns (half the clock period, to satisfy setup time for the pins before the next clock edge). Using these values, the equation becomes $2.5 = 5(1 - e^{-10ns/(20pf * R)})$, which gives R = 721Ω. This raises the next question, can the pads sink that much current? That size resistor would require the pads to sink 5/721 = 6.93mA in order to pull the wire down to a logic 0. The NFET in the pad has an effective W/L of 90 and K = 140.6 $\mu A/V^2$. The lowest

current sinking ability of the pad will occur toward the low end of the output voltage swing when the NFET is in linear mode. In this region, we have:

$$I_{ds} = \beta \bullet \left[(V_{gs} - V_t) \bullet V_{ds} - \frac{Vds^2}{2} \right]$$

Solving for Ids = 6.93 mA, gives Vds = 0.139 V. This is well below the threshold for a logic 0. The only other concern is the power and ground pins then; in particular, the ground pins on the chip. That current has to be sunk through the communications control pins and then out the ground pins. There are six ports on the chip, meaning that there are 6*4 = 24 of these pins that could all be pulled down simultaneously. This shouldn't happen very often because the quiescent operating state of the pins leaves them in the logic 1 state, but, as a worst case, the total current being drained out through the ground pins would be 24*6.93mA = 166.4mA. This, spread across 5 ground pins, gives 33.28mA per ground pin. The rule of thumb for a 20 year lifetime is 1 mA per micron of wire.

A.5 Mesh

As discussed in Chapter 4, the mesh supports permanent nearest neighbor connections in the four cardinal directions between IFUs. The Skip Bus segments link neighboring units in similar fashion. At the east and west, edges are connected so that it is as if the edges are in fact nearest neighbors. Two inputs to the mesh come from the crossbar to the top of each column for a total of eight, one input enters through the local north connection for the top IFU and the other enters through the northern Skip Bus terminal for the top IFU. Note that programming information can only propagate through the local connections within the mesh and a stream

whose data is to enter through the Skip Bus terminal must end at the top of the mesh. Based on the assumption that some degree of data combination/reduction will occur within the mesh, and in an effort to reduce the size of the crossbar, only one output runs from the bottom of each column in the mesh back to the crossbar, for a total of four. The Skip Bus connections leaving the bottom of the mesh are dead ends and have been terminated with PFET pull ups. These could still be used for one of the "back tracking" connection paths through the IFU as discussed below.
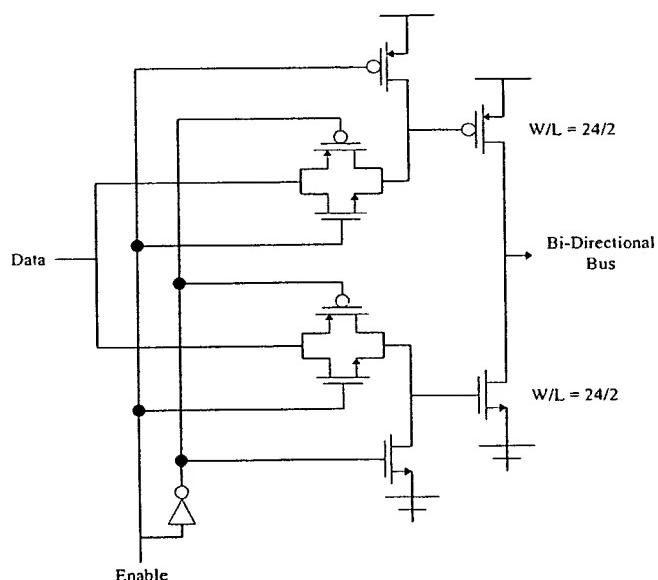
Figure A.16 – High Strength Bi-Directional Driver.

To ensure that the propagation delay times between the east and west edges were comparable to those between other parts of the mesh, the signal drivers used were enhanced. As part of the enhancement, the Skip Bus segment drivers were redesigned. Within the normal mesh connections, complementary transmission gates were used at the outputs of normal inverters to

provide the tri-state functionality needed to support bi-directional connectivity. Originally, NFET pass transistors were used instead of fully complementary transmission gates; however, the resultantly degraded signals are connected to large BDS drivers. When the weak logic 1 voltage was used to drive the BDS, large leakage currents were encountered. The overall static power consumption was deemed to great and so complementary transmission gates were used. Because of the long wire lengths needed to create the wrap-around connections, the bi-directional drivers were upgraded to the circuit shown in Figure A.16. The driving transistors have been upgraded to BDS drivers, giving an approximate W/L ratio of 12. Further, the output is no longer funneled through a pass transistor, thus reducing the resistance along the signal path and increasing overall speed. Simulations show that the new design drives the wrap-around signals faster than the interconnections within the perimeter of the mesh. Of course, this is at the expense of added complexity.

A.6 Interconnected Functional Unit
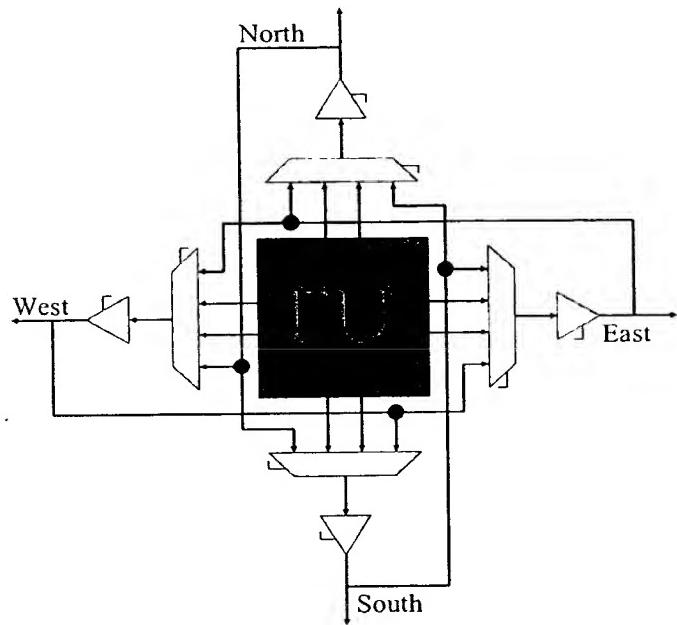
Figure A.17 - IFU Connectivity.

A.6.1 Data Path

The basic design of the Interconnected Functional Unit (IFU) is shown in Figure A.17. The signal paths shown are for the Skip Bus only as the Local Bus or nearest neighbor connections have not been shown for the sake of clarity. This structure is used for the normal data path, and once for each of the flags: shift (FS), carry (FC) and conditional (FN). Note that the Skip Bus connectivity within an IFU does not support arbitrary connections between segments that are adjacent to the FU. In order to reduce circuit complexity, only certain connections are directly supported. Any given segment can be connected to one of four different

329

input sources: FU Bus output, FU Aux output, the Skip Bus segment to the compass opposite of this one or the Skip Bus segment to the compass right of this one. The FU Bus and Aux outputs are derived from values computed by the Functional Unit. The segment to the compass opposite for the west Skip Bus segment would be the east segment and its compass right would be the north segment. These can be directly connected to create an un-buffered signal path so that data may skip over the IFU with no delays. In this way, long signal paths may be created in the mesh that have single clock cycle latencies.

The direction of data flow through a given segment is controlled by the configuration information of the FUs. Since two adjacent FUs must agree on the direction of signal propagation through a given segment, the directional control signals are distributed between them. An FU only contains the configuration bits that control the Skip Bus segments that are to its east and south. A signal line runs between an FU to the one adjacent on each side that sets the direction to be used for the shared segment.

It is possible to create other connections by "back tracking" the signal along the skip bus segments. For example, a path from west to north may be established by programming the eastern segment to take the value of the western one and then programming the north to take the east. The flow of signals shown was chosen with computation in mind, especially shifting operations found in numeric computation. In many numeric algorithms, such as integer multiplication and floating point addition, shifting to the left is often used. Thus, it seemed reasonable to include a direct signal path from the north to the west segment. Also, the ability to pass over a unit entirely in either the vertical or horizontal directions seemed prudent. From there the design was mirrored around all four sides of the unit for ease of layout and design.

330

The input multiplexers for the FU flags and registers are contained within the IFU. There are separate multiplexers for the *Left* and *Right Input Registers* and one for each of the three flags. Data from eight different values may be selected as an input source, these include the local connections from the north, south, east and west directions as well as Skip Bus segments from all four directions. Each multiplexer itself is implemented as a partial ring running around the outside of the FU. Where a given signal path comes into the IFU, NFET pass transistors are used to provide a tri-state switch that can be turned on to allow that path to drive the ring. A 3:8 decoder is used to drive the control lines for the switches; thus, guaranteeing that only one input source will drive the ring at any given time. In addition, a special loop-back path is included from the FU Bus output back to the *Left Input Register*. This is used during configuration and it can also be used during normal processing to provide a convenient feedback path for use in constructing an accumulator. This is connected to the same ring-like structure as the normal NFET tri-state devices for the 8:1 multiplexer.

A.6.2 Programming

Programming information can arrive at the IFU at any time from any of the four cardinal directions along the local mesh buses. There are four configuration control signals, one associated with each direction: *ProgramInN, ProgramInE, ProgramInS* and *ProgramInW*. When new configuration data arrives from a given direction, its programming signal will be asserted, causing the IFU to enter programming mode. The IFU state machine latches the programming signals arriving from all four directions and holds them until the end of the configuration header has been reached. This guarantees that once stream selection has occurred, a second stream

331

attempting to program the IFU will not be able to succeed until after programming by the first stream has been completed. At first this may seem superfluous, and that such a condition could be considered programmer error and so disallowed; however, when broadcast programming is occurring it is possible for an FU to be programmed to send the stream header in several directions, and then to have one of those branches arrive back at the originating FU before programming is complete. In this case, the FU must ignore the returning branch so that the original programming stream will not be ignored or overridden because of the arrival of the new stream.
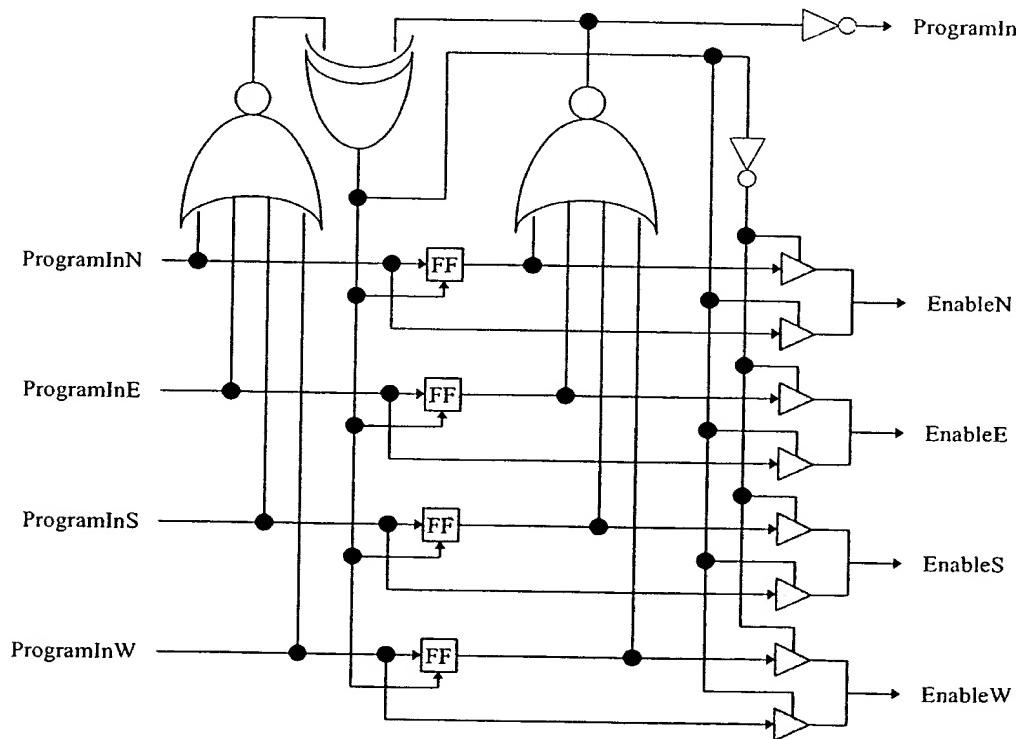
Figure A.18 - IFU State Machine Schematic.

Figure A.18 shows how this function is performed. The four *ProgramInX* signals shown on the left are the programming signals associated with the data words arriving on the local connections from the four cardinal directions. When a word is flagged as being programming information (part of a stream header), the corresponding signal will be asserted (logic 1). The four *EnableX* signals are the overrides generated by the circuit that signify that programming information should be accepted from that direction. The *ProgramIn* signal goes to the FU to signify that the word currently latched into the *Right Input Register* is configuration information. Note that this signal is delayed for one clock pulse, just as the data is delayed one clock pulse as it is latched into the *Right Input Register*.

Initially, all four programming signals are zero and the flip flops are initialized to zero. This being the case, both NOR gates will output a logic 1 and the XOR will output a logic 0; hence, the flip flop load signals will be logic 0 and the *ProgramIn* signal will be a logic 0, also the override *EnableX* signals will all be zero. During some future clock cycle, one or more of the *ProgramInX* signals will be asserted. These signal(s) will be immediately forwarded through to the corresponding *EnableX* signals so that the data can be steered to the FU. At the same time, the flip flop load signals will be asserted, latching the new states of the *ProgramInX* signals. Once the asserted *ProgramInX* signals have been latched, the values held in the flip flops are sent out as the *EnableX* values instead of the *ProgramInX* signals. Also, the load signal will be deasserted until all four *ProgramInX* lines go to a logic 0 value. The flip flops will not latch any changes in the state of the *ProgramInX* lines until no programming information is arriving from any direction. Thus, when broadcast programming is used, the FU will receive the initial stream, lock down the direction it came from, send out new streams, and if any of them come back from

333

a different direction, they will be ignored. Because they are ignored, the *EnableX* outputs will remain stable for the duration of programming, so that the IFU will remain locked onto the same configuration source the entire time.

Since configuration information enters the FU through the *Right Input Register*, the 8:1 multiplexer controlling the data source for that register must be forced to select the configuration information as the new data source. This is accomplished by the *EnableX* signals from the IFU state machine driving a prioritized set of pull down transistors on the inputs to the 3:8 decoder that is used to select the input source for the *Right Input Register*. The pull downs must be prioritized because of the case in which configuration information arrives at the IFU from two different directions simultaneously; such as could happen during broadcast programming. In this case, multiple *EnableX* signals could be asserted so that the decoder logic must uniquely select one direction instead of selecting some muddled combination of the two.
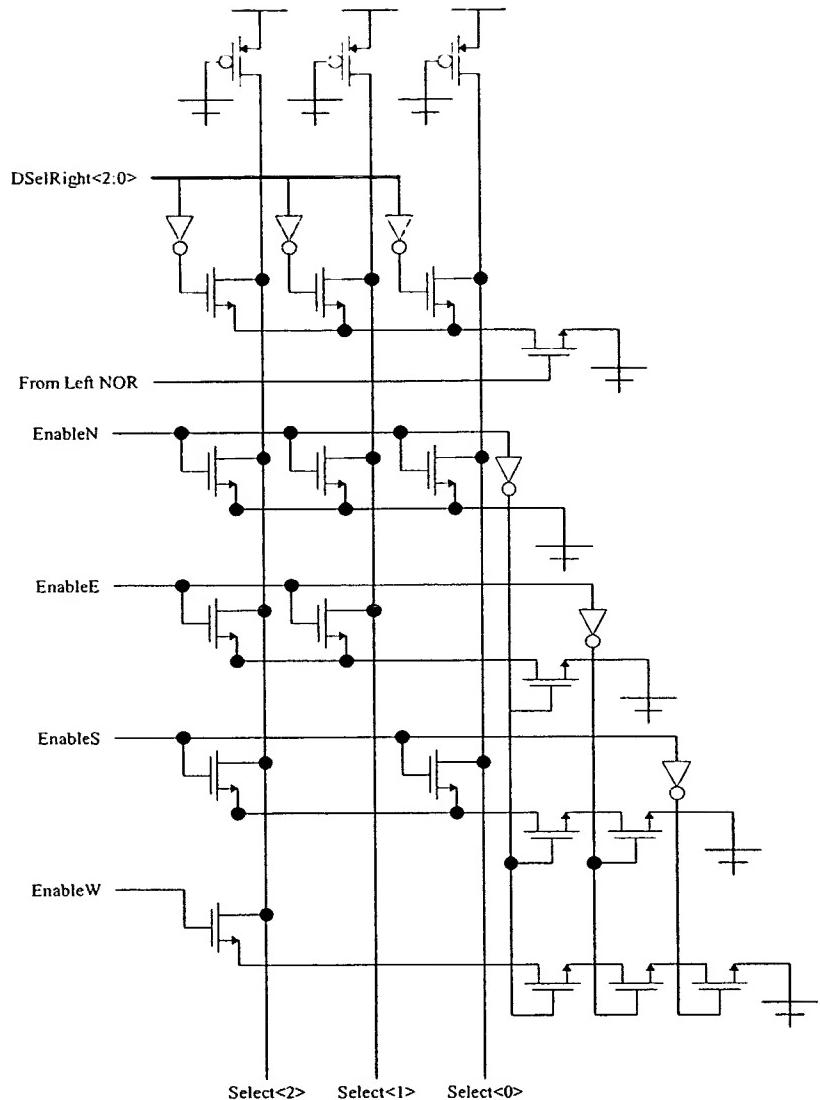
Figure A.19 - IFU Right Input Register Priority Encoder.

The priority encoder circuit used is depicted in Figure A.19. The priority assignment, in

descending order of importance, is: *EnableN*, *EnableE*, *EnableS* and lowest priority is assigned

to *EnableW*. The decoder used to drive the enable lines for the *Right Input Register* should be connected to the *Select<2:0>* lines shown at the bottom.

Table A.6 - IFU Input Select Encoding.

Select Address	Direction Selected
000	Local North
001	Local East
010	Local South
011	Local West
100	Skip North
101	Skip East
110	Skip South
111	Skip West

The binary values for selecting from the eight possible input sources are given in Table A.6. The signal labeled *"From Left NOR"* originates at the output of the NOR gate on the left hand side of Figure A.18 in the IFU state machine. It will be high when none of the *ProgramIn*X signals is high, meaning that no programming information is being received. In this situation, the NFET that this signal drives will be turned on and the select signals specified in the FU configuration (*DSelRight<2:0>*) will drive the decoder lines to the appropriate value. When any of the *ProgramIn*X signals entering the IFU state machine is asserted, it will cause the NOR gate output to go low, thus disabling the normal configuration. When the *Enable*X line for a particular direction is asserted, the normal configuration is disabled and the appropriate lines are pulled low to select that direction. Notice that the pull down transistors for *Enable*X lines further down the tree cannot make a complete circuit to ground unless all *Enable*X lines above them are at a logic 0. This is achieved through the use of the horizontal NFETs on the right, and it

336

guarantees that only one of the four *EnableX* lines will be able to pull the decoder address lines low at any one time.

The direction of stream propagation through the mesh is controlled by four bits in the first word of an FU configuration packet, one bit for each direction of propagation: north, south, east and west. If the bit for a given direction is set, then all configuration information in the stream header following the packet for the current FU will be forwarded along that direction. Note that it is possible to have multiple bits set and; thus, the stream can be made to diverge at a given point in the mesh and proceed along two or more different paths. Note that reconfiguring along the reverse path of propagation is disallowed by the hardware as this is considered to be programmer error.

A.6.3 Electrical Design

The design of the rings and driver placement was carefully considered to balance propagation delay time versus area consumed by the drivers. The Skip Bus was particularly interesting because it needs to have a low delay path so that it can skip over many IFUs in a single clock period without being slowed down by excessive capacitive loads.
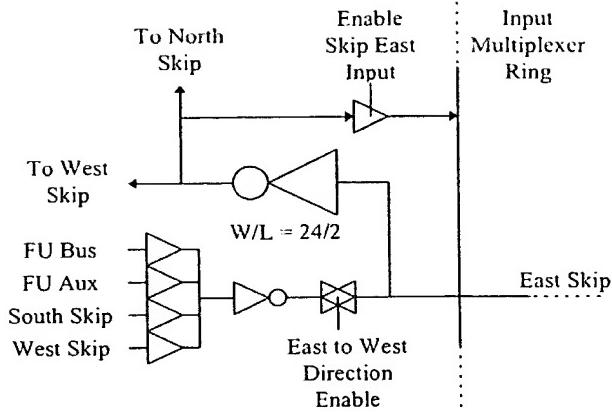
Figure A.20 - Detail of East Skip Bus Design.

The construction of the eastern Skip Bus circuitry leaving an IFU that is internal to the mesh is shown in Figure A.20. The *East Skip* line shown would be connected to the western Skip Bus connection on this IFU's counterpart to the right. The *Input Multiplexer Ring* is part of the 8:1 multiplexer used to select the data source for flags and data for this IFU. One of the NFET tri-state constructions is shown near the top of the figure, controlled by *Enable Skip East Input*. The controlling signal originates from a 3:8 decoder that is located elsewhere. Near the bottom is one of the fully complementary transmission gates used to control the direction of the east going Skip Bus. A complementary control signal is fed to the IFU to the east so that the Skip Bus will only be driven from one side at a time. The four input multiplexer is driven by the *FU Bus* and *Aux* outputs as well as the compass opposite and compass right Skip Bus segments; relative to the eastern one. The capacitance of the Skip Bus segment itself is relatively low, while the capacitance of the multiplexer ring is rather high. Thus, minimum size inverters are used to drive the Skip Bus segment, but the segment itself is used to drive high strength inverters in both this and the IFU to the east. The output of these high strength inverters is used to charge

338

the multiplexer ring and to drive the eastern Skip Bus signal to the compass left and compass opposite multiplexers. Though it may seem desirable to isolate the Skip Bus signal path entirely from the path used to charge the multiplexer ring, simulations show that the NFET pass transistor in the signal path to the ring provides enough resistance to prevent the large load of the ring from significantly degrading the propagation time along the continuing Skip Bus paths.

A.7 Functional Unit

The design of the FU was a continuing compromise between implementation size, the addition of computational features and speed of operation. It was decided early on that the critical path of performance through the system should always be through the FU data path. Thus, every addition to the functionality of the FU directly affected the maximum clock rate at which the Colt could be run. Great care was taken to balance the potential benefits of a design addition versus the impact it would have on the clock rate. In particular, the design of the ALU and the shifter both reflect tuning and simulation effort to achieve the lowest propagation times possible without greatly increasing the size of implementation.

A.7.1 Data Path
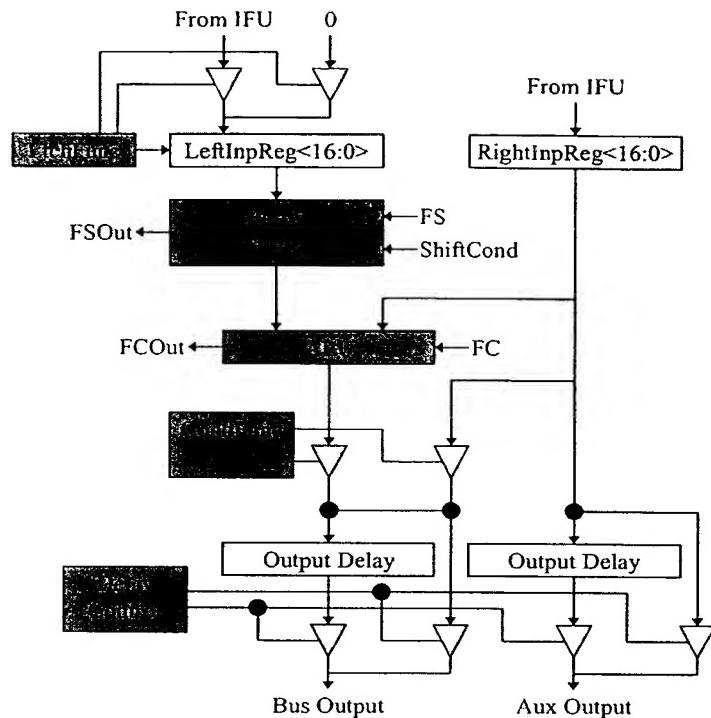
Figure A.21 - Detailed FU Data Path.

A more detailed schematic of the FU data path is given in Figure A.21. This shows some of the control flags and the units that they affect as the two special purpose logic functions used to control the operation of the *Conditional Unit* and the *LeftInpReg<16:0>* loading function. In reality more control is needed than is shown here, but that is mainly related to the configuration process and will be explained in Section 5.

340

A.7.1.1 Left Input Register Latch Function

The *Left Input Register* latch function is intended to allow the programmer flexibility in storing a value for repeated use in a series of computations. There are four basic modes of operation.

Table A.7 - Left Input Register Latch Function.

Select	Function
0	Latch All Values
1	If FN = 1 Load Valid Zero, Else Load Only Valid Data
2	Load Only Valid Data
3	Never Load, Use Configuration Constant

Table A.7 gives selection indexes and functions for each mode. In Mode 0, all data is loaded into the *Left Input Register*, regardless of whether it is valid or the value of the *FN* flag. Mode 2 sets the register to only load data that has the valid bit set. This could be useful for some accumulator type applications where valid data is interspersed with invalid data due to flow control problems, non-deterministic execution times or for other reasons. Mode 3 stops the register from ever latching data, thus causing the programmer defined value for that register to remain in place. Such a value could be used as an additive constant or as a logical mask for the value that is loaded into the *Right Input Register*. Mode 1 is the most complicated in that when $FN = 0$ it behaves just as Mode 2 does, but, when $FN = 1$, Mode 1 causes a valid zero to be loaded into the *Left Input Register*. This mode can be used to create a resetable accumulator using a single FU in combination with the loop back data path to the *Left Input Register*; the reset of which is controlled by the *FN* flag. The *Right Input Register* always loads a new value at the beginning of the clock period, just as the *Left Input Register* will if it is in Mode 0.

```
# Num. Vars, Num Bin Vars., Size Of Multi Value Vars
.mv 7 6 3

# Name parts of multi value var.
.label var 6 Load Mux0 Mux1

# Mux0 loads data, Mux1 loads a valid 0
.ilb Lf1 Lf0 P Pr Vb Fn

# Specifies that we will give ON-set, OFF-set and DC-set
.type fdr

# Programming Modes
--1--- 110

# Latch Everything 00
000--- 110

# Load Valid Zero If Conditional Flag Set 01
# Otherwise Load Only Valid Data
010--1 101
010-10 110
010-00 000

# Load Data If Valid Bit Set 10
100-1- 110
100-0- 000

# Never Load Data 11
110--- 000
110--- 000
```

Figure A.22 - Left Input Register Latch Function PLA File.

As for the data port state machine, *misII* [83] was used to minimize the set of special

purpose logic functions needed to control the input multiplexer and load signal of the *Left Input*

Register. The multiplexer chooses between the value selected by the IFU for the *Left Input*

Register and the bit pattern needed to load a valid zero. As shown in the PLA file, the *Mux0*

output selects the normal IFU value and the *Mux1* output selects the valid zero value. The *Load*

function, when high, causes the *Left Input Register* to latch the value present at its inputs.

Table A.8 - Latch Function Output Equations.

Output	Equation
Mux0	$LF_1' * Vb * FN' + LF_0' * Vb + LF_1' * LF_0' + P$
Mux1	$LF_1' * LF_0 * P' * FN$
Load	$Mux1 + Mux0$

342

Table A.8 gives the resulting output equations, where the *LF* signals are the high and low bits of

the select number, *Vb* is the valid bit of the incoming data item from the IFU, *FN* is the *FN* flag

and *P* is the *ProgramClock* signal generated by the FU state machine.

A.7.1.2 Barrel Shifter
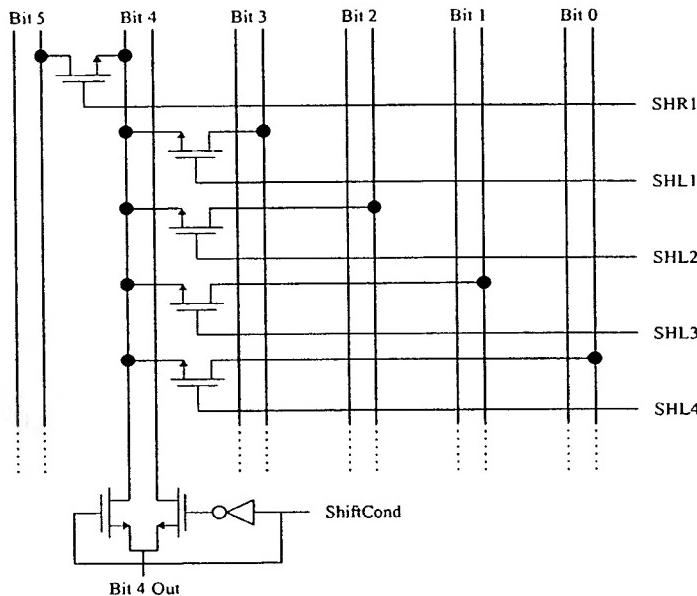
Figure A.23 - Barrel Shifter Construction.

The barrel shifter was again a compromise between size, speed and flexibility. The

design that was chosen is shown in Figure A.23. Only the circuitry entering the bit 4 position is

shown. Two vertical wires are used for each bit, the one to the right of the label carries the

original value of the bit and the one to the left of the label carries the shifted value of the bit. At

the bottom of the structure, two NFET pass transistors are used to select which of the two signals

will be propagated for that bit, based on the value of *ShiftCond*. A 3:5 decoder drives the shift

343

operation select lines, labeled *SHR1*, *SHL1*, *SHL2*, *SHL3* and *SHL4*, thus guaranteeing the mutually exclusive access to the shifted value line. The data signals go through at most two NFET pass transistors, making the circuit very fast. Also, the shift function select lines, driven by the 3:5 decoder need not transition quickly since the control signals are set by the configuration data and do not change again until the next configuration. This was exploited by using small drivers for these lines.

Not shown are the lines running from the other bits to their shifted value lines (to the left of the bit label), but they look similar to the ones entering the bit 4 position. When shifting by a single bit in either direction the bit shifted in is taken from the *FS* flag. When shifting by more than one bit to the left, the bits shifted in are taken from the top bits of *Right Input Register*. The *FSOut* flag is equal to the bit that would be shifted into the bit 16 position when shifting to the left, or is equal to the bit shifted out the right hand side when shifting to the right.

Table A.9 - Shift Condition Input Select.

Shift Condition Select	Input Source
0	0
1	FN

Table A.9 shows that the *ShiftCond* signal can be driven by either a constant 0 or by the *FN* flag. Thus, the shifter can be set to never function by selecting input 0, and it can also be set to always shift using the same setting because there is an optional inversion after the input multiplexer, making the actual value of *ShiftCond* a logic 1. The optional inversion can also be applied to the *FN* flag, of course.

A.7.1.3 ALU
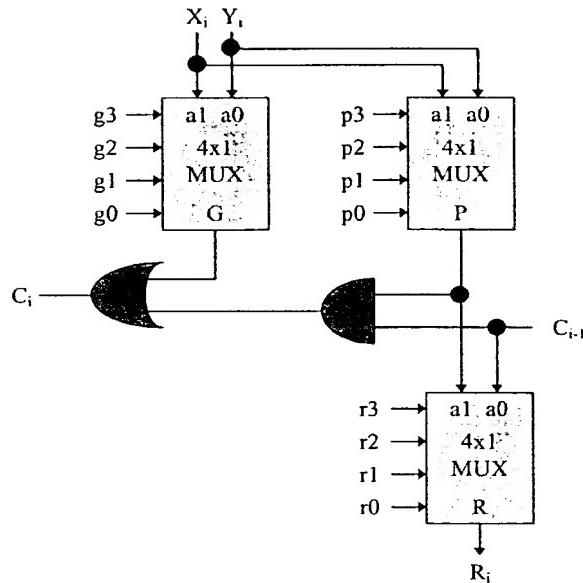
Figure A.24 - General ALU Bit Slice.

The critical path of the entire chip is defined by the timing of the Arithmetic Logic Unit. Because of this, a great deal of effort was made at both the schematic and layout levels to minimize the worst case delay through the carry path of the unit. Delay in VLSI is largely due to the capacitive load on a given signal. This circuit was designed to minimize the capacitance on the carry signal as it propagated through the 16-bit ALU. At the same time, it is important that the reconfigurable nature of the unit be maintained so that it can be applied to a wide variety of problems, implementing various operators. Because of the need for flexibility, a carry look ahead or some similar scheme was ruled out. Satisfaction of the goal was achieved using an ALU based on the general Propagate, Generate, Result scheme was employed, as has been discussed.

345

The general form for such a unit is repeated here in Figure A.24. The three functions P, G and R are defined by 12 configuration bits from the FU; the same set of bits are applied to all 16 bits of the ALU. By using different bit combinations, many different functions can be realized, including all Boolean functions of two variables, and also many mathematical functions such as add, subtract, increment, decrement, two's complement, shift left, etc. The bit pattern to perform an addition using the circuit above is $P = 6$, $G = 8$ and $R = 6$, for example. Though the figure above is useful to consider when programming the unit, it is not an optimal implementation of the bit slice from a VLSI standpoint.
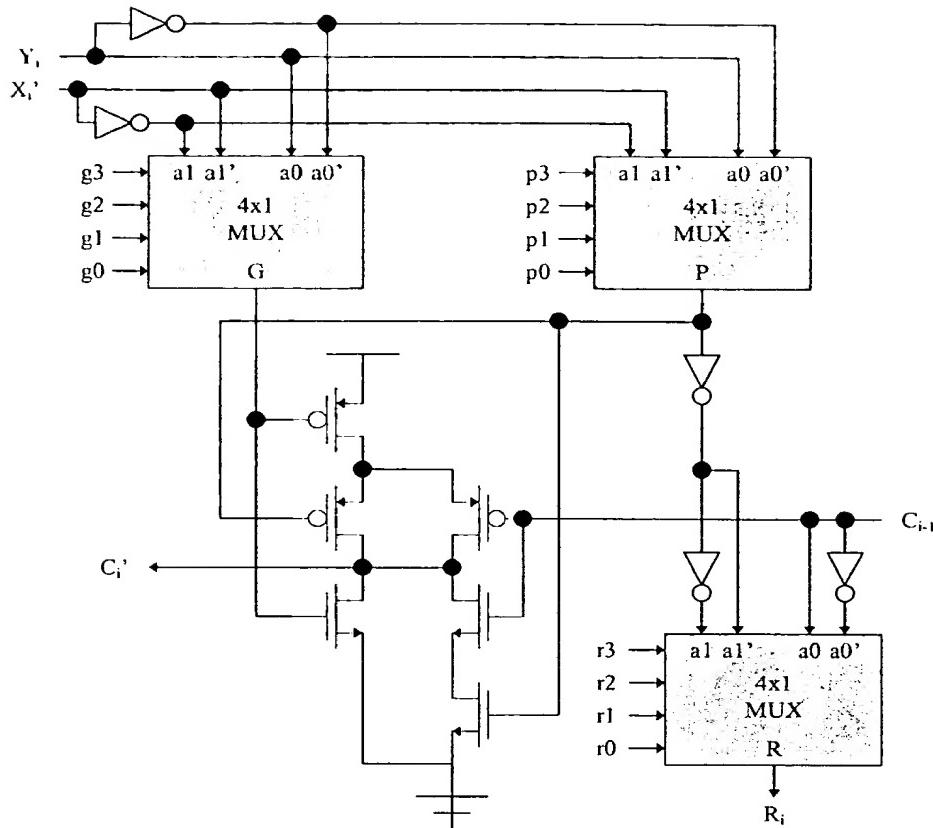
Figure A.25 - Even ALU Bit Slice.

Figure A.25 depicts the circuitry used for even bit numbers (0,2,4,...) within the ALU.

Each of the multiplexers consists of six NFET pass transistors whose gates are driven by the four

addressing signals shown entering the device. Note that the output of the P function is inverted

before it drives the addressing lines for the R function. This is because the NFET pass transistors

degrade logic 1 signals to approximately 3.6V using a supply of 5V at normal circuit speeds. If

the degraded signal then drives the gate of another NFET pass transistor the new Vgs is even

lower, thus degrading a logic 1 value again to a value of approximately 2.2V. This is

347

unacceptably low so a restoring inverter is used to raise the voltage output of the P multiplexer function back up to 5V before it drives the R multiplexer addressing lines. Also notice that the true form of the X input does not enter the circuit, but rather the inverted form. This was done for the same reason, though it isn't as obvious. The X input is driven by the output of the barrel shifter, which is another NFET pass transistor circuit. Thus, the X input must be boosted back to original signal strength before it can drive the P and G multiplexer addressing lines. The Y input comes straight from the *Left Input Register* and so does not suffer from this problem.
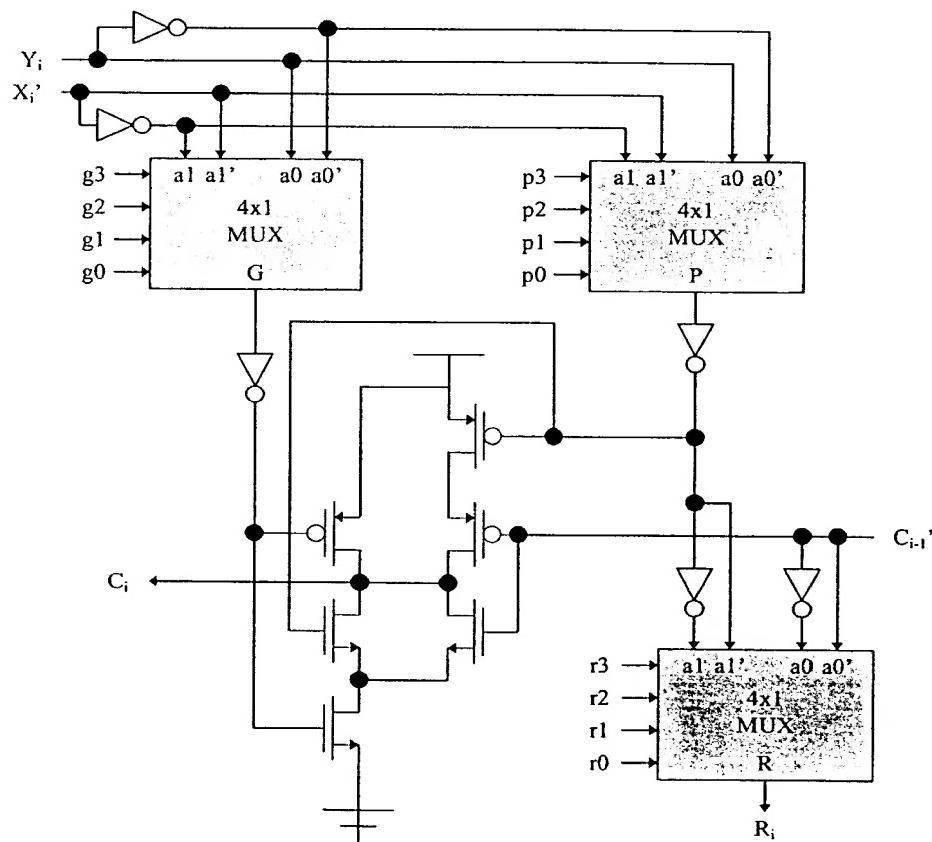
Figure A.26 - Odd ALU Bit Slice.

Figure A.26 shows the bit slice used for odd numbered bits within the ALU (1,3,5,...). It is much the same as the even numbered slice and the same programming bits are used to drive the *P*, *G* and *R* functions. The differences between the two is the sense of the carry in and carry out that are expected. The even numbered slice expects the true form of the carry in and gives an inverted carry out. The odd numbered slice takes an inverted carry in and returns the true form of carry out. Thus when the two are strung together to form a complete ALU they complement each other well. The load on the carry line was reduced by using this scheme of alternation because it neatly avoids the need for an inverter to restore the sense of the carry bit before proceeding with computation. The circuit was shown to be very fast by simulations and it can perform an addition in less than 7 ns.

A.7.1.4 Conditional Select Function

The conditional unit allows the programmer to conditionally select between the value produced by the ALU and the value currently in the *Right Input Register* based on the *FN* flag. The *Conditional Mode* bit in the FU configuration controls the behavior of the function. If *CondMode* is a 0, then the value of the ALU is always chosen, regardless of the value of *FN*. If *CondMode* is a 1, then the selection depends on *FN*. If *FN* is a logic 1, the value produced by the ALU is passed through. If the *FN* flag is a logic 0, the value in the *Right Input Register* is passed through.

This function has a number of uses, for example, it can be used to merge two streams containing valid and invalid data to produce a single stream of valid data. This situation could occur when performing conditional execution when both sides of a conditional must be executed.

After computation, either one or the other of the two sides of the conditional will be flagged invalid for each result. These two streams then need to be merged so that only the valid results remain. The *Conditional Unit* can perform this function by sending one stream to the *Left Input Register* and the other to the *Right Input Register*. Then, by setting the *FNOut* flag to point to the valid bit of the *Right Input Register* (*RightInpReg<16>*), and setting the *FN* flag to point to the *FNOut* flag the *Conditional Unit* can perform the merge operation.

The unit also has applicability for conditional operations where the operation performed by the ALU may be switched in and out. A conditional increment by two is a good example. The constant two could be programmed into the *Left Input Register* and the data stream could be directed into the *Right Input Register*, with the ALU configured to perform addition. With the *FN* flag controlled by the source of the conditional, the increment by two can then be selected or neglected as desired.

```
# Num. Vars, Num Bin Vars., Size Of Multi Value Vars
.mv 4 3 2

# Name parts of multi value var.
.label var=3 ALU Right

.ilb P Cm Fn

# Specifies that we will give ON-set, OFF-set and DC-set
.type fdr

# Programming Mode
1-- 01

# Unconditional ALU
00- 10

# Conditional ALU or Right Operand
011 10
010 01
```

Figure A.27 - Conditional Unit PLA File.

The combinational logic required to control the two enable signals for the *Conditional Unit* was again derived using misII. The PLA file used is shown in Figure A.27. Note that the operation of the function is dependent on three variables: *P* (*ProgramIn*), *Cm* (*Conditional*

350

Mode) and the *FN* flag. The *ProgramIn* signal is true whenever programming information is being injected into the FU. After this FU has been configured, the remainder of the stream header must be forwarded to the rest of the chip. The path used to pass the stream header through the computational section of the FU stems from the *Right Input Register*, then bypassing the ALU through the *Conditional Unit*, around the *Optional Output Delay* and out the bottom of the computational path. Thus, whenever programming is occurring, the function of the *Conditional Unit* is overridden so that the *Right Input Register* is always selected as the output.

Table A.10 – Conditional Unit Control Equations.

Output	Equation
ALU	RightInpReg'
RightInpReg	ConditionMode * FN' + ProgramIn

Table A.10 gives the equations governing the control of the two select signals leaving the function.

A.7.1.5 Control Flags

There are three user-controllable flags that can affect the computation performed by the FU: the shift flag (*FS*), the carry flag (*FC*) and the conditional flag (*FN*). There are three versions of each of these, one representing the raw value received from the network through the IFU (*FSIn*, *FCIn* and *FNIn*), the second is the value actually applied to the control circuitry of the FU (*FS*, *FC* and *FN*), and the third is the value that is output from this unit to the network (*FSOut*, *FCOut* and *FNOut*). Multiplexers allow the programmer to select the data source for the

351

flags actually applied to the units of the FU (*FS*, *FC* and *FN*) and an optional input inversion allows the programmer to select either the true or complemented form of that signal.

Table A.11 - Control Flag Multiplexer Values.

Select	FS	FC	FN
00	0	0	0
01	FSIn	FCIn (Latched)	FNIn
10	CondOutput<15>	FSOut	FSOut
11	FNOut	FN	FNOut

Table A.11 shows the possible input sources that can be selected for each of the flags *FS*, *FC* and *FN*. There are several important details to be noted from the table. First, all of the "In" versions of the flags as received from the IFU are inverted from the form that was originally generated. This is a peculiarity of the way in which the signal drivers were implemented, but it is in no way debilitating since there is an optional inversion at the output of each of the multiplexers described in Table A.11. Thus, the true or inverted forms of any of the signals can easily be selected, allowing the constant 0 shown along the top row to actually drive a constant 1 as well.

Another interesting point is that the *FCIn* selection for the *FC* signal is latched. This was intentionally inserted in order to prevent propagation delay problems when stringing multiple FUs together to form a long accumulator. The critical path of the circuit goes directly through the carry chain of the ALU; thus, if the ALUs from two FUs were connected together to form a 32-bit adder, for example, the critical path would lengthen and the clock would have to be slowed. By latching the *FCIn* signal from the previous FU in the chain, the operation is effectively pipelined; avoiding added propagation delay and allowing such a configuration to

operate at full speed. The drawback is that operands being processed through the chained ALU structure must then be skewed by one clock cycle just as is done for the carry signal, but this can be performed using either the optional output delay of the FU, skewing of the data in the streams, or some other means. The other possible inputs for the *FC* flag are not latched since they are assumed to be generated from a relatively local source (such as within the same FU) and thus would not significantly affect the critical path. In practice, this assumption has been found to be less than valid and improvements in the next generation chip will need to be made.

A number of "Out" signals are also listed in Table A.11. *CondOutput<15>* is the sign bit of the result leaving the conditional multiplexer of the FU and has been included for certain mathematical operations. *FSOut* is the bit shifted out of either the left or right hand side of the barrel shifter when a shift by 1 is performed in either direction. When shifting left by more than a single bit, *FSOut* is the bit that would be shifted into the bit 16 position. Note that it is possible for the *FN* signal to be routed through to the *FC* signal, this allows the full functionality of the *FN* bit to be used to calculate the carry in. This path also allows a logical loop to be formed through the *FNOut* flag, and this situation should be avoided in practice.

Table A.12 - FNOut Input Sources.

Select	Input Source
0	Right Input Register <0>
1	Right Input Register <1>
2	Right Input Register <2>
3	Right Input Register <3>
4	Right Input Register <4>
5	ALU Output <15>
6	FCOut
7	FNIn
8	NOR(Right Input Register <15:14>)
9	NOR(Right Input Register <15:13>)
10	Right Input Register <15>
11	Right Input Register <16>
12	NOR(Right Input Register <15:12>)
13	ALU Overflow

The *FNOut* flag is intended to allow a great deal of flexibility in selecting signals throughout the FU for use in controlling computations. The possible input sources that can be selected for the *FNOut* signal are shown in Table A.12. Many of these have been chosen for specific computational problems; however, they have general applicability as well. The first five possible select values take as inputs the lower five bits of the value currently latched in the *Right Input Register*. These were included specifically to decode binary values by bit position so that weighted operations could be performed. For example, using these bits the fixed offset shifting operations performed by the barrel shifter can be pipelined to form a unit capable of variably shifting by any value between 0 and 31. Such a unit would be useful in the mantissa alignment process that must occur before performing a floating point addition.
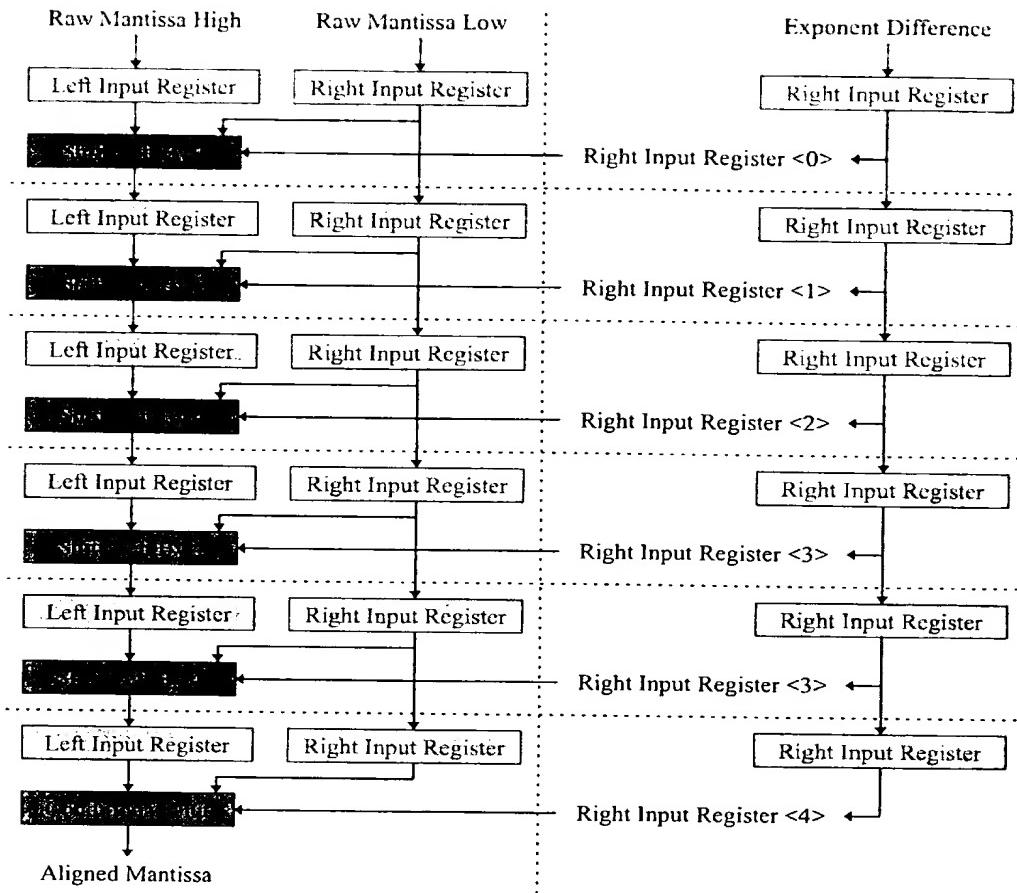
Figure A.28 - Floating Point Mantissa Alignment.

To perform such an alignment, the difference between the exponents of the two values is taken and is then latched into the *Right Input Registers* of successive FUs, as shown on the right of Figure A.28. The *Left* and *Right Input Registers* of the column of FUs shown on the left could be configured to latch the upper and lower bits of the mantissa of the smaller floating point value, which would then be shifted by successive amounts as dictated by the lower bit positions of the exponent difference. The boundaries between successive FUs is shown by dotted lines in the

figure. Note that shift left by eight is not directly supported by the Barrel Shifter and so this must be implemented by two shift left by four operations. Including direct support for shift left by eight in the barrel shifter would have eased the implementation of this function, but such an inclusion would have doubled the size of the shifter and so was considered to expensive. Neither is the shift left by 16 operation supported by the barrel shifter; however, this function can be simulated by using the *Conditional Unit* to choose the *Left Input Register* if no shifting is to be performed, or the *Right Input Register* if a shift left by 16 is required.
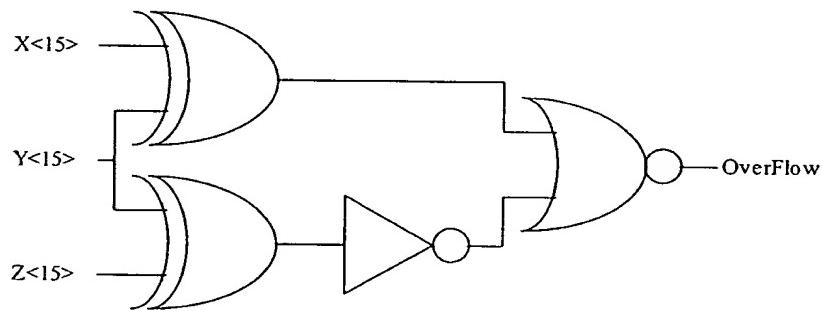
Figure A.29 - ALU Overflow Function.

Table A.12 shows some other possible values for *FNOut*, including the sign bits for the ALU and *Right Input Register* (bit 15) and the valid bit of the *Right Input Register* (bit 16). Obviously, the ability to select the sign bit allows computation to proceed based on whether the result is positive or negative, both at the input stage and after the ALU operation. The *ALUOverflow* function shown in Figure A.29 automatically detects a two's complement arithmetic overflow from the ALU operation. There are several ways to implement such a detector circuit, but this arrangement was used due to the relative accessibility of the various signals in the layout. *X<15>* and *Y<15>* are the sign bits of the two input operands to the ALU and *Z<15>* is the sign bit of the result. The value of *ALUOverflow* will be a logic 1 if an

356

arithmetic overflow is detected and logic 0 otherwise. This function was included at the last minute as a result of mapping the floating point multiplier to the Colt architecture. It was realized then that such a function would be very useful and small to implement as a special purpose function within the FU while configuring a detector by other means would have been resource intensive. As more such functions are found in the future, they can be added to the input list for *FNOut*, or the equivalent circuit.

The various NOR functions shown have been included to aid in floating point mantissa normalization. By using a strategy similar to that shown for mantissa alignment, a series of FUs can be cascaded together to shift a given mantissa left until the leading digit is a 1. First, index 12, the NOR of *Right Input Register* bits 15:12, would be selected along with the shift left by four function of the barrel shifter. This signal can be routed to *FN* and inverted before being used so that it is actually the OR of those bits, and hence, if it is a 1 no shift should be performed. If the OR is a 0, then a shift by four should be done in an effort to get a leading 1 in bit 15. After a series of these operations, the NOR of *Right Input Register* bits 15:13 could be tested, following by the NOR of bits 15:14 and finally, just bit 15 would need to be tested. The same flag that is used to control the shift condition could be used to control a conditional decrement operation in another FU so that the exponent would be properly adjusted at the same time.

In all of the possible input sources for *FNOut*, a high concentration was placed on the *Right Input Register* while the *Left Input Register* has been ignored. This was justified because both the *Left* and *Right Input Registers* can independently select from the same set of eight input sources through the IFU. Thus, the value to be used for the condition being tested can always be latched into the *Right Input Register*. The only restraint that favoritism of one register over the

357

other causes, then, is in cases where the operation desired and the conditional value required for that operation must be in separate paths. To reduce this restraint, the conditions were concentrated on the *Right Input Register* since the *Left Input Register* has the added dependence of the barrel shifter following it. Ideally, all inputs would be available from either register, but in order to minimize excess hardware, and to minimize the number of configuration bits required for the FU, this was not done. Thus far in practice the choice of concentration of the conditional flags in the *Right Input Register* has not hindered optimal implementation.

A.7.2 Programming Path
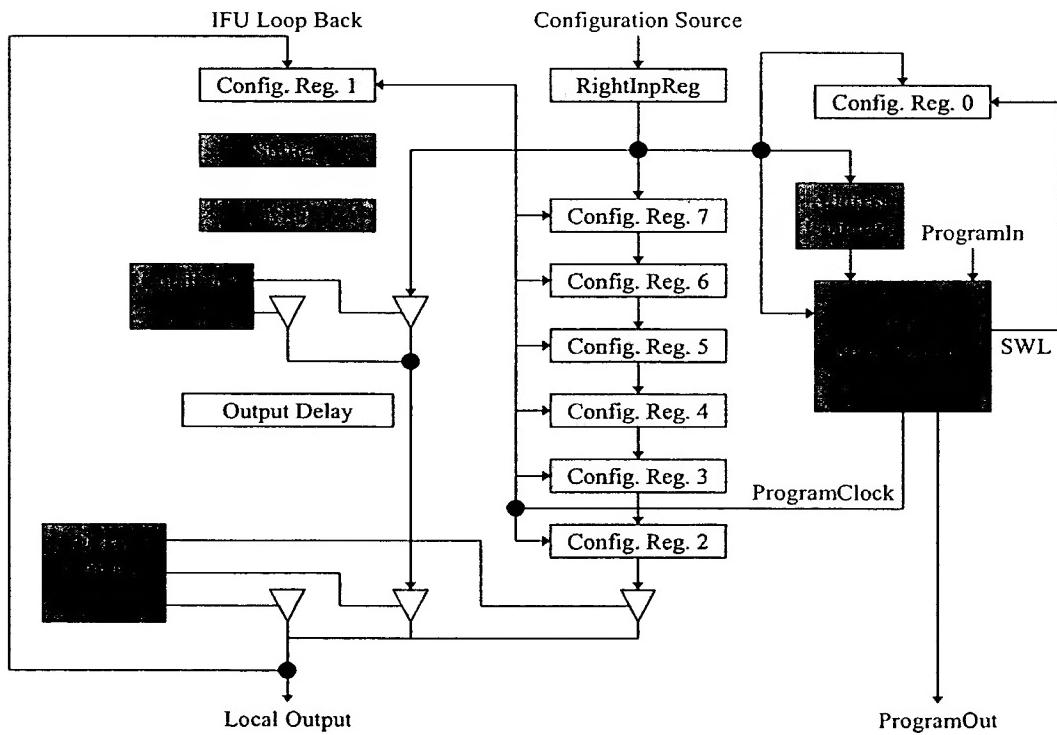
Figure A.30 - FU Configuration Path.

An overview of the path through the FU for configuration data is shown in Figure A.30. The FU state machine on the right hand side orchestrates the configuration process. As discussed for the IFU, when a new stream header arrives on one of the four local bus connections, it overrides current IFU and FU operations. The IFU directs the stream header to the *Right Input Register* of the FU and the *ProgramIn* signal is asserted. The FU latches the first word into the *Right Input Register* and checks several signals. The lower six bits of the word are run through the *Address Comparator* for this unit and will be true if the address matches. The *StartOfPacketMarker* bit is checked to ensure that this is the first word of a new packet. Also, the valid bit for the word and the *ProgramIn* signal are checked. If all these are asserted, the FU state machine begins the configuration process.

First, the remaining bits of the first configuration word are latched into *Configuration Register 0*, which is only eight bits wide. The other bits of the first word were used for the unit address, *StartOfPacketMarker* and *SingleWordProgram* and are no longer needed. These eight bits contain the four bits controlling the stream forwarding direction and several other bits. During the following seven cycles, the *ProgramClock* signal is asserted by the FU state machine so that the configuration registers will latch in the remaining configuration words for the FU; eight in all. Only 15 bits of configuration information are contained in each of the remaining seven words since bit 15 is used as the *StartOfPacketMarker* flag and must be 0 for all of these. The configuration registers are numbered in the order that their configuration words appear in the programming packet; hence, *Configuration Register 1* (actually the *Left Input Register*) takes the lower 15 bits of its programmed value from the second word in the programming packet. However, notice that *Configuration Registers 2* through *7* will all temporarily latch its value as

359

the words from the packet are shifted through them. The *Delay Control* circuit is designed to drive the value of *Configuration Register 2* onto the FU Bus Output while *ProgramClock* is asserted. Also, the IFU *Bus Output* loop-back to the *Left Input Register* is automatically enabled while *ProgramClock* is asserted. Between the two functions, a path is established from *Configuration Register 2* back to the *Left Input Register* (*Configuration Register 1*). Finally, the *ProgramClock* signal overrides the *Left Input Register* latch function so that it will latch the configuration word.

When the end of the FU configuration packet has been reached, a new packet will be latched into the *Right Input Register* that will have the *StartOfPacketMarker* set. This cues the FU state machine to deassert the *ProgramClock* signal and begin passing the rest of the stream header through the FU untouched. Though *ProgramClock* is then driven to a logic 0, the *ProgramIn* signal from the IFU is still at a logic 1 since the remainder of the stream header does contain programming information. This causes the *Delay Control* circuit to switch the active path to that of the *Output Delay* bypass. Also, the *ProgramIn* signal overrides the *Conditional Unit* so that the ALU bypass is also in effect, creating a clear path from the *Right Input Register* straight through to the *Bus Output*. From there the configuration information is sent to the IFU and dispersed in all directions indicated by the configuration information. Note that the *ProgramOut* signal is also asserted for this data as well so that it will be recognized as new configuration information by the next unit(s) along the programming path.

360

A.7.3 FU State Machine

Input Vector: VPAMS
V = Valid Bit
P = Program Bit
A = Address Compare
M = Start Of Packet Marker
S = Single Word FU Packet

Normal
Active

11110
001

11111
001

11X0X   0XXXX
110XX   10XXX
010   000

Program

11X1X
010

Program
Active

X0XXX
000

01XXX
000

11X0X
100

X0XXX
000

X1XXX
010

Output Vector: PcPoSwl
Pc = Program Clock (AH)
Po = Program Out (AH)
Swl = Single Word Latch (AH)

State	Assignment
Normal Active	00
Program	10
Program Active	11

Figure A.31 – FU State Machine Transition Diagram.

The state transition diagram for the FU is shown in Figure A.31 along with the state

assignment used. The Mealy machine has 5 inputs, all of which are associated with the word

currently latched into the *Right Input Register*: the valid bit in position 16 (V), the *ProgramIn*

signal from the IFU (P), the result of checking the lower six bits versus the address comparator

(A), the *StartOfPacketMarker* in bit 15 (M) and the *SingleWordProgram* bit in position 14 (S).

The purpose of the FU state machine is to control the configuration process. As with the other

units on the Colt, the FU is reset into the *Normal Active* state and remains there until a word arrives that has the *Program* signal set (signified by *ProgramIn* at this level). If a word arrives that is configuration information ($P=1$), is valid ($V=1$), has the right address ($A=1$) and is the first word in a new packet ($M=1$) the state machine will transition to one of two different programming states. Note that the address comparators are designed to respond to either the specific address for this unit or to the broadcast address (111111) so that broadcast programming is possible.

The *SingleWordProgram* bit (S) determines the state transition if all the conditions above are met. This bit was included to allow an existing configuration to be left intact and a stream header to be forwarded through the FU using only a single configuration word. This is useful for reconfiguring a single unit along a previously existing path, for example. If the S bit is set, then the state machine will transition directly to the *Program Active* state; however, it will also assert the *Single Word Latch* (*SWL*) signal so that *Configuration Register 0* will latch the new programming word. Because *Configuration Register 0* latches the word, it is possible to reconfigured those bits using this word. They include the bits used to control the stream forwarding direction and so it is possible to change the path of the programming stream from its original course through the chip.

If the *SingleWordProgram* bit (S) is not set, then all eight configuration registers in the FU will be reprogrammed as the state machine will transition to, and operate in, the *Program* state. Normally, the state machine remains in the *Program* state until a word is latched into the *RightInputRegister* for which the *StartOfPacketMarker* is set. At this point, the state machine will stop latching words into the configuration registers of the FU, transition to the *Program*

362

Active state and begin forwarding the remainder of the stream header to the next unit(s) by asserting the *ProgramOut* signal.

The intent of the *Program Active* state is to put the FU into a state in which it will ignore all further programming information until the entirety of the remainder of the stream header has been forwarded and *Normal Active* mode has been resumed. This allows following packets to contain any information, even addresses that would otherwise match that of this FU, without being intercepted by the state machine of this FU. The advantages of this mode for unit and chip chaining have already been discussed.

```
# Num. Vars, Num Bin Vars., Size Of Multi Value Vars
.mv 8 7 5
# Name parts of multi value var.
.label var=7 Q1p Q2p Pc Po Sw1

# Names of binary variables
.ilb Q1 Q2 V P A M S

# Specifies that we will give ON-set, OFF-set and DC-set
.type fdr

# Normal Active 00
000---- 00000
0010--- 00000
0011-0- 00000
00110-- 00000
0011111 11001
0011110 10001

# Program 10
1001--- 10000
1011-0- 10100
10-0--- 00000
1011-1- 11010

# Program Active 11
11-1--- 11010
11-0--- 00000

# Not Used 01
01----- -----
```

Figure A.32 – FU State Machine PLA File.

Once the end of the stream header has been reached, the FU will transition back to the *Normal Active* state and will behave as dictated by the configuration information that was latched during programming. It will remain in the *Normal Active* state until new configuration information arrives. Once again, misII was used to generate the next state and output equations

for the state machine. The PLA file is shown in and the equations generated are shown in Table

A.13.

Table A.13 - FU State Machine Equations.

Function	Equation
Q_1^+	$P\ Q_1 + \{Swl\}$
Q_2^+	$S\ \{Swl\} + \{Po\}$
Pc	$P\ Q_1\ V\ \{Po\}'$
Po	$M\ P\ Q_1\ V + P\ Q_1\ Q_2$
Swl	$A\ M\ P\ Q_1'\ Q_2'\ V$

364

B. FP Multiplier & Tool Overview

In an effort to document the various tools that were developed to aid in the configuration and testing of Colt, the various steps needed to configure the floating point multiplier shown in Figure 5.7 will be given here. The tools that are used are: coltgui, dfc, mergestl and pwl. These deal with successively finer degrees of detail in order to abstract the architectural quirks of Colt. However, these tools provide little more than an "assembly language" equivalent interface to the Colt CCM and improved tools will be the subject of future research.
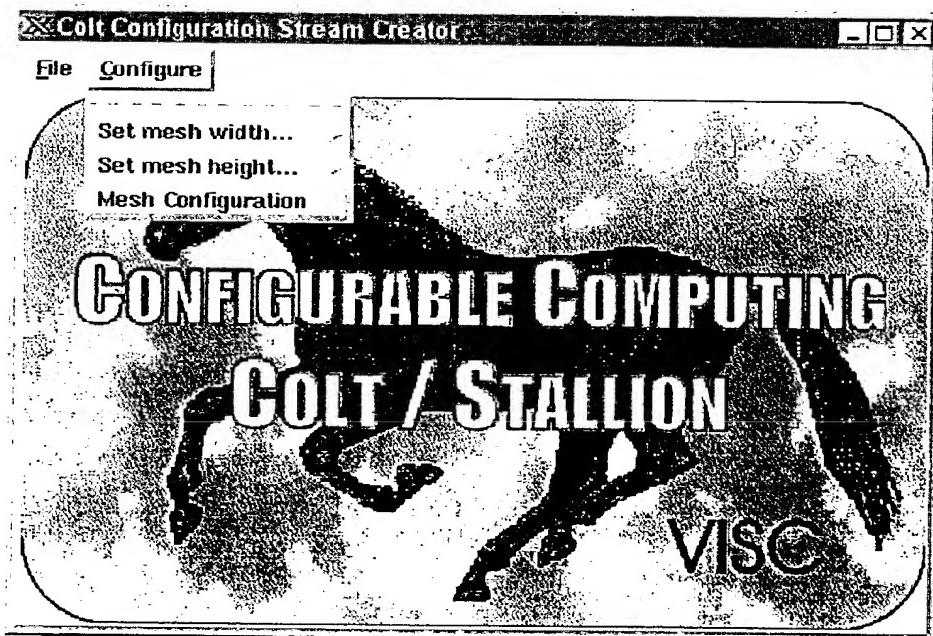
Figure B.1 - Coltgui Initial Window.

Coltgui was written by Dr. Athanas in TK/TCL to run under the Unix X-Windows environment. It allows the user to specify a configuration for Colt using a system of menu items that toggle and set the values of the various control bits and multiplexer inputs for the IFU, data ports and crossbar, simply by selecting menu options.
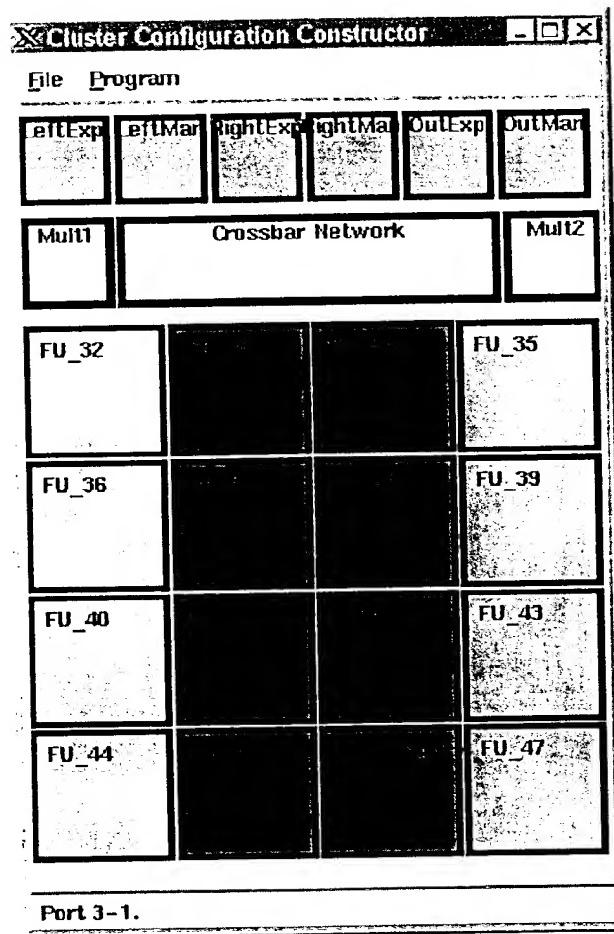
Figure B.2 - Coltgui Mesh Configuration Window.

The actual configuration process is done using the window shown above. Here the 4x4 mesh can clearly be seen, and the three colors delineate the three stream paths used to configure

366

the floating point multiplier. Notice that coltgui allows the user to specify names for the various units and also their colors for ease of use. The current version also shows two multipliers on the chip, but in fact only one is present. In fact, although the multipliers are shown for completeness, no configuration information is needed to configure the multiplier, other than routing the input streams through the two input words via the crossbar.
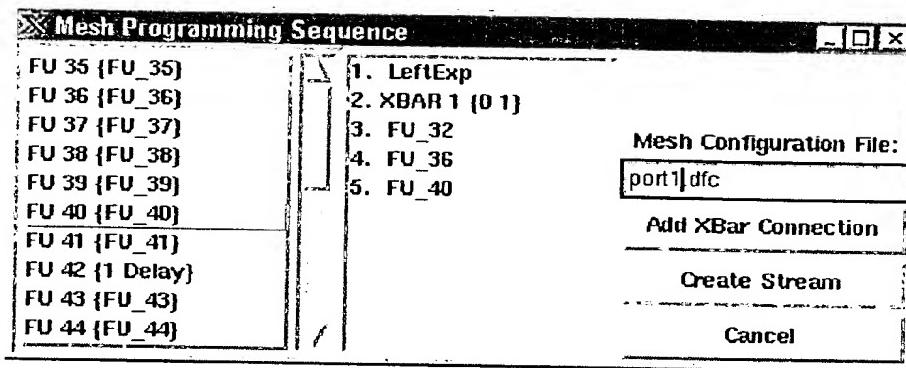
Figure B.3 – Coltgui Stream Sequencing Window.

When the configuration for each IFU and data port has been completed, the order of the configuration packets for each stream must be specified using the window shown above. The configuration being shown is for the stream that enters Data Port 1, goes through the crossbar and enters the local north input of the left column of the mesh.

367
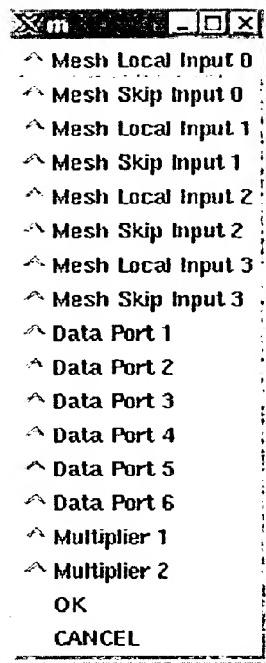
Figure B.4 – Coltgui Crossbar Control Window.

Crossbar configurations are a simple matter of specifying output addresses from the crossbar. The window for programming these addresses is shown above. Once the configuration order has been entered, the designer presses the *Create Stream* button and a text file is written to disk.

This file contains dfc (data flow compiler) code. Dfc was written by Ray Bittner as a layer of abstraction between the specification of data port, crossbar and IFU configuration information and the specification of the actual bit stream. Each dfc file contains the configuration information for each unit along a path through Colt. The configuration information can be specified using user defined labels or using the numeric or direct binary values needed to configure the units.

368

```
#include "fielddef.h"

; Default FU To Use For Configuration
DeclareFU <Name>
        ...
EndFU

; Stream Header To Inject Into Port 1
Path [0]
        DataPort
                ...
        EndDataPort
        CrossBar
                ...
        EndCrossBar
        FU <DeclareFU>
                ...
        EndFU
EndPath
EndProgram
```

Figure B.5 - General DFC Program Format.

The general form of a dfc path is shown above. The path is started, followed by a series

of data port, crossbar and FU blocks, given in the order that they should be configured by the

stream header. The FU body is unique in that parts of the configuration used can be inherited

from a previously defined skeletal body as defined by the *DeclareFU* block. When the name

given after the *FU* declaration matches the name specified after a *DeclareFU* block, the

configuration information given by the *DeclareFU* block will be used to fill in the fields for that

FU. This allows the programmer to specify a generic adder, for example, without specifying the

exact routing of source operands. All fields that are not specified in the *DeclareFU* block must

be filled in by the programmer for an inheriting *FU* block.

All the fields that need to be assigned a value are given by the file fielddef.h. This file

contains the field definitions and possible value labels for each configurable field for each unit

type. The fielddef.h file needed to configure Colt is listed here:

```
; The number shows the width in bits of the whole field
Field SingleWordProgram[1]
        ; This is in the form [ConfigWord,HighBit:LowBit] ...
```

```
        Position [0,14]
        Label Enable 1
        Label Disable 0
EndField

Field ProgramNorth[1]
        Position [0,13]
        Label Enable 1
        Label Disable 0
EndField

Field ProgramEast[1]
        Position [0,12]
        Label Enable 1
        Label Disable 0
EndField

Field ProgramSouth[1]
        Position [0,11]
        Label Enable 1
        Label Disable 0
EndField

Field ProgramWest[1]
        Position [0,10]
        Label Enable 1
        Label Disable 0
EndField

Field LeftOperand[17]
        Position [0,9:8] [1,14:0]
EndField

Field ValidBitMux[2]
        Position [0,7:6]
        Label FN 00
        Label FNAndLeftInpReg16 01
        Label FNAndRightInpReg16 10
        Label LeftInpReg16AndRightInpReg16 11
EndField

Field Address[6]
        Position [0,5:0]
EndField

;***************************************************************
; ConfigReg2 Stuff
;***************************************************************
Field FNSelMuxW[2]
        Position [2,13] [2,14]
        Label SkipN 00
        Label SkipE 01
        Label FNOut 10
EndField

Field FNSelMuxS[2]
        Position [2,11] [2,12]
```

370

```
          Label SkipW 00
          Label SkipN 01
          Label FNOut 10
EndField

Field FNSelMuxN[2]
          Position [2,9] [2,10]
          Label SkipE 00
          Label SkipS 01
          Label FNOut 10
EndField

Field FNSelMuxE[2]
          Position [2,7] [2,8]
          Label SkipS 00
          Label SkipW 01
          Label FNOut 10
EndField

Field FNSel[3]
             Position [2,4] [2,5] [2,6]
          Label LocalN      000
          Label LocalE      001
          Label LocalS      010
          Label LocalW      011
          Label SkipN 100
          Label SkipE 101
          Label SkipS 110
          Label SkipW 111
EndField

Field DSelRight[3]
             Position [2,3:1]
          Label LocalN      000
          Label LocalE      001
          Label LocalS      010
          Label LocalW      011
          Label SkipN 100
          Label SkipE 101
          Label SkipS 110
          Label SkipW 111
EndField

;****************************************************************
; ConfigReg3 Stuff
;****************************************************************
Field FCInpSel[2]
          Position [3,9] [3,14]
          Label Zero 00
          Label FCIn 01
          Label FSOut 10
          Label FN 11
EndField

Field FNInv[1]
          Position [3,13]
          Label Enable 1
```

```
            Label Disable 0
      EndField

      Field OutputDelay[1]
            Position [3,12]
            Label Enable 1
            Label Disable 0
      EndField

      Field CondMode[1]
            Position [3,11]
            Label ALU 0
            Label ALUOrRightOperand 1
      EndField

      Field FCInv[1]
            Position [3,10]
            Label Enable 1
            Label Disable 0
      EndField

      Field DSelLeft[3]
            Position [3,8:7] [2,0]
            Label LocalN         000
            Label LocalE         001
            Label LocalS         010
            Label LocalW         011
            Label SkipN 100
            Label SkipE 101
            Label SkipS 110
            Label SkipW 111
      EndField

      Field FCSelMuxN[2]
            Position [3,6:5]
            Label SkipE 00
            Label SkipS 01
            Label FCOut 10
      EndField

      Field FCSelMuxW[2]
            Position [3,4:3]
            Label SkipN 00
            Label SkipE 01
            Label FCOut 10
      EndField

      Field FCSelMuxS[2]
            Position [3,1] [3,2]
            Label SkipW 00
            Label SkipN 01
            Label FCOut 10
      EndField


;*********************************************************************
; ConfigReg4 Stuff
```

372

```
;*****************************************************************
Field FSInv[1]
        Position [4,12]
        Label Enable 1
        Label Disable 0
EndField

Field FSInpSel[2]
        Position [4,11:10]
        Label Zero 00
        Label FSIn 01
        Label CondSign 10
        Label FNOut 11
EndField

Field FNInpSel[2]
        Position [4,8] [4,9]
        Label Zero 00
        Label FNIn 01
        Label FSOut 10
        Label FNOut 11
EndField

Field FCSelMuxE[2]
        Position [4,7] [3,0]
        Label SkipS 00
        Label SkipW 01
        Label FCOut 10
EndField

Field FCMuxDirNS[1]
        Position [4,6]
        Label North 1
        Label South 0
EndField

Field FCSel[3]
        Position [4,3] [4,4] [4,5]
        Label LocalN     000
        Label LocalE     001
        Label LocalS     010
        Label LocalW     011
        Label SkipN 100
        Label SkipE 101
        Label SkipS 110
        Label SkipW 111
EndField

Field DSelMuxS[2]
        Position [4,1] [4,2]
        Label SkipW 00
        Label FUBusOut   01
        Label FUAuxOut   10
        Label SkipN 11
EndField

Field FNMuxDirNS[1]
```

373

```
        Position [4,0]
        Label North 1
        Label South 0
EndField


;******************************************************************
; ConfigReg5 Stuff
;******************************************************************
Field R[4]
        Position [5,10]  [5,11]  [5,12]  [5,13]
        Label Add  0110
EndField

Field FNOutSel[4]
        Position [4,13]  [4,14]  [5,8]  [5,9]
        Label RightOperandBit0   0000
        Label RightOperandBit1   0001
        Label RightOperandBit2   0010
        Label RightOperandBit3   0011
        Label RightOperandBit4   0100
        Label ALUSign            0101
        Label FCOut       0110
        Label FNIn        0111
        Label NorRO15_14  1000
        Label NorRO15_13  1001
        Label RightOperandSign   1010
        Label RightOperandValid 1011
        Label NorRO15_12  1100
        Label ALUOverFlow 1101

EndField

Field DMuxDirNS[1]
        Position [5,7]
        Label North 1
        Label South 0
EndField

Field FCMuxDirEW[1]
        Position [5,6]
        Label East 1
        Label West 0
EndField

Field FNMuxDirEW[1]
        Position [5,5]
        Label East 1
        Label West 0
EndField

Field LeftInputSelect[1]
        Position [5,4]
        Label Normal 0
        Label LoopBack 1
EndField
```

374

```
Field DSelMuxE[2]
      Position [5,2] [5,3]
      Label SkipS 00
      Label FUBusOut    01
      Label FUAuxOut    10
      Label SkipW 11
EndField

Field DMuxDirEW[1]
      Position [5,1]
      Label East 1
      Label West 0
EndField

Field FSMuxDirEW[1]
      Position [5,0]
      Label East 1
      Label West 0
EndField



;****************************************************************
; ConfigReg6 Stuff
;****************************************************************
Field P[4]
      Position [6,11] [6,12] [6,13] [6,14]
      Label Add 0110
EndField

Field G[4]
        Position [5,14] [6,8] [6,9] [6,10]
      Label Add 1000
EndField

Field DSelMuxN[2]
      Position [6,7:6]
      Label SkipE 00
      Label FUBusOut    01
      Label FUAuxOut    10
      Label SkipS 11
EndField

Field FSSelMuxE[2]
      Position [6,5:4]
      Label SkipS 00
      Label SkipW 01
      Label FSOut 10
EndField

Field FSSelMuxN[2]
      Position [6,3:2]
      Label SkipE 00
      Label SkipS 01
      Label FSOut 10
EndField
```

```
Field FSSelMuxW[2]
      Position [6,0]  [6,1]
      Label SkipN 00
      Label SkipE 01
      Label FSOut 10
EndField


;******************************************************************
; ConfigReg7 Stuff
;******************************************************************
Field ShiftType[3]
      Position [7,14:12]
      Label ShiftLeft1 000
      Label ShiftLeft2 001
      Label ShiftLeft3 010
      Label ShiftLeft4 011
      Label ShiftRight1 100
EndField

Field ShiftCondInv[1]
      Position [7,11]
      Label Enable 1
      Label Disable 0
EndField

Field ShiftCondInpSel[1]
      Position [7,10]
      Label Zero 0
      Label FN 1
EndField

Field Lf[2]
      Position [7,8]  [7,9]
      Label LoadAll 00
      Label LoadZero 01
      Label LoadValid 10
      Label LoadNone 11
EndField

Field FSSelMuxS[2]
      Position [7,6]  [7,7]
      Label SkipW 00
      Label SkipN 01
      Label FSOut 10
EndField

Field FSMuxDirNS[1]
      Position [7,5]
      Label South 1
      Label North 0
EndField

Field DSelMuxW[2]
      Position [7,3]  [7,4]
      Label SkipN 00
```

```
            Label FUBusOut      01
            Label FUAuxOut      10
            Label SkipE 11
EndField

Field FSSel[3]
            Position [7,0] [7,1] [7,2]
            Label LocalN        000
            Label LocalE        001
            Label LocalS        010
            Label LocalW        011
            Label SkipN 100
            Label SkipE 101
            Label SkipS 110
            Label SkipW 111
EndField




;****************************************************************
; DataPort Stuff
;****************************************************************
Field DPLoopBit[1]
            Position [0,13]
                Label Disable 0
                Label Enable 1
EndField

Field DPRWBit[1]
            Position [0,14]
                Label Read 0
                Label Write 1
EndField

Field DPSyncBit[1]
            Position [0,12]
                Label Sync 1
                Label NoSync 0
EndField

Field DPSyncMP6[1]
            Position [0,11]
EndField

Field DPSyncMP5[1]
            Position [0,10]
EndField

Field DPSyncMP4[1]
            Position [0,9]
EndField

Field DPSyncMP3[1]
            Position [0,8]
EndField

Field DPSyncMP2[1]
```

377

```
            Position [0,7]
EndField

Field DPSyncMP1[1]
        Position [0,6]
EndField

Field DPAddress[6]
        Position [0,5:0]
EndField




;*********************************************************************
; Cross Bar Stuff
;*********************************************************************
Field XBExtra[9]
        Position [0,14:6]
            Label Null 000000000
EndField

Field XBAddress[6]    .
        Position [0,5:0]
EndField
```

The file consists of a series of field definitions; one for each configurable field on the Colt CCM. First, a *Field* label is given, followed by the width in bits of the field that it names. Then the *Position* of the field within the configuration registers for the unit is shown. Field bits that are scattered throughout the configuration registers can be specified by using multiple bit ranges after the *Position* keyword. Finally, a list of labels that can be assigned to the field are given. The label name indicates the function to be configured and a binary value after the name gives the bits to be programmed into the field to cause that function to be performed.

When dfc is run, each unit's configuration is read and translated, using the fielddef.h file, into bit vector(s) suitable for inclusion in a Colt stream header. These are packets of 16-bit values conforming to the packet format previously discussed. If a given bit position in the configuration registers is assigned to more than once by overlapping fields, or multiple

assignments to the same field, dfc will flag the error and give some indication as to which fields are causing the problem. This is a common error when a *DeclareFU* is specified and then used to derive another FU. The programmer may inadvertently assign a value to a field that had been previously assigned by the *DeclareFU* and cause this error. Various other error conditions are also detected.

The dfc files generated by coltgui for the floating point multiplier are given below. The first listing gives the dfc code for the stream header to be injected into *Data Port 1*.

```
;
; Stream file created by COLTGUI, version code 841847636
; Created on Mon Sep 16 15:48:24 EDT 1996 by user bittner
;
#include "fielddef.h"
Path [0]
;
; Configuration for Port 1-1
;
        DataPort
                DPAddress = 24
                DPRWBit = 0
                DPSyncBit = 1
                DPSyncMP1 = 1
                DPSyncMP2 = 0
                DPSyncMP3 = 0
                DPSyncMP4 = 0
                DPSyncMP5 = 0
                DPSyncMP6 = 0
                DPLoopBit = 0
        EndDataPort
;
; Configuration for crossbar, Address = 1
;
        CrossBar
                Address = 1
                XBExtra = 0
        EndCrossBar
;
; Configuration for FU FU_32, Address = 32
;
        FU
                Address = 32
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 0
                ShiftCondInv = 1
                ShiftCondInpSel = 0
                LeftOperand = 0
```

```
                    OutputDelay = 0
                    FCInpSel = 0
                    FCInv = 0
                    FNInpSel = 0
                    FNInv = 0
                    FSInpSel = 0
                    FSInv = 0
                    CondMode = 0
                    FNOutSel = 0
                    LeftInputSelect = 0
                    Lf = 0
                    P = 12
                    G = 0
                    R = 12
                    ProgramNorth = 0
                    ProgramEast = 0
                    ProgramSouth = 1
                    ProgramWest = 0
                    DSelLeft = 0
                    DSelRight = 0
                    DMuxDirNS = 0
                    DMuxDirEW = 1
                    DSelMuxN = 0
                    DSelMuxE = 1
                    DSelMuxS = 0
                    DSelMuxW = 0
                    FCSel = 0
                    FCMuxDirNS = 0
                    FCMuxDirEW = 0
                    FCSelMuxN = 0
                    FCSelMuxE = 0
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 0
                    FSMuxDirNS = 1
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 2
                    FSSelMuxW = 0
                    FNSel = 0
                    FNMuxDirNS = 0
                    FNMuxDirEW = 0
                    FNSelMuxN = 0
                    FNSelMuxE = 0
                    FNSelMuxS = 0
                    FNSelMuxW = 0
            EndFU
    ;
    ; Configuration for FU FU_36, Address = 36
    ;
            FU
                    Address = 36
                    SingleWordProgram = 0
                    ValidBitMux = 1
                    ShiftType = 0
                    ShiftCondInv = 1
```

380

```
                    ShiftCondInpSel = 0
                    LeftOperand = 65536
                    OutputDelay = 1
                    FCInpSel = 0
                    FCInv = 0
                    FNInpSel = 0
                    FNInv = 1
                    FSInpSel = 1
                    FSInv = 1
                    CondMode = 0
                    FNOutSel = 0
                    LeftInputSelect = 0
                    Lf = 3
                    P = 12
                    G = 0
                    R = 12
                    ProgramNorth = 0
                    ProgramEast = 0
                    ProgramSouth = 1
                    ProgramWest = 0
                    DSelLeft = 0
                    DSelRight = 0
                    DMuxDirNS = 1
                    DMuxDirEW = 1
                    DSelMuxN = 0
                    DSelMuxE = 0
                    DSelMuxS = 0
                    DSelMuxW = 0
                    FCSel = 0
                    FCMuxDirNS = 0
                    FCMuxDirEW = 0
                    FCSelMuxN = 0
                    FCSelMuxE = 0
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 5
                    FSMuxDirNS = 1
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 1
                    FSSelMuxW = 0
                    FNSel = 0
                    FNMuxDirNS = 0
                    FNMuxDirEW = 0
                    FNSelMuxN = 0
                    FNSelMuxE = 0
                    FNSelMuxS = 0
                    FNSelMuxW = 0
            EndFU
    ;
    ; Configuration for FU FU_40, Address = 40
    ;
            FU
                    Address = 40
                    SingleWordProgram = 0
                    ValidBitMux = 3
```

381

```
                ShiftType = 0
                ShiftCondInv = 0
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 1
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 0
                FSInv = 0
                CondMode = 0
                FNOutSel = 0
                LeftInputSelect = 0
                Lf = 0
                P = 6
                G = 8
                R = 6
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 2
                DSelRight = 0
                DMuxDirNS = 0
                DMuxDirEW = 1
                DSelMuxN = 0
                DSelMuxE = 1
                DSelMuxS = 1
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 0
                FSSel = 0
                FSMuxDirNS = 1
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 1
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 0
        EndFU
;
; Configuration for FU FU_44, Address = 44
;
        FU
                Address = 44
```

382

```
                        SingleWordProgram - 0
                        ValidBitMux = 1
                        ShiftType = 0
                        ShiftCondInv - 1
                        ShiftCondInpSel = 0
                        LeftOperand = 65536
                        OutputDelay - 1
                        FCInpSel = 0
                        FCInv = 0
                        FNInpSel = 0
                        FNInv = 1
                        FSInpSel = 1
                        FSInv = 1
                        CondMode = 0
                        FNOutSel = 0
                        LeftInputSelect = 0
                        Lf = 3
                        P = 12
                        G = 0
                        R = 12
                        ProgramNorth = 0
                        ProgramEast = 0
                        ProgramSouth = 0
                        ProgramWest = 0
                        DSelLeft = 0
                        DSelRight = 0
                        DMuxDirNS = 1
                        DMuxDirEW = 1
                        DSelMuxN = 0
                        DSelMuxE = 0
                        DSelMuxS = 0
                        DSelMuxW = 0
                        FCSel = 0
                        FCMuxDirNS = 0
                        FCMuxDirEW = 0
                        FCSelMuxN = 0
                        FCSelMuxE = 0
                        FCSelMuxS = 0
                        FCSelMuxW = 0
                        FSSel = 4
                        FSMuxDirNS = 0
                        FSMuxDirEW = 0
                        FSSelMuxN = 0
                        FSSelMuxE = 0
                        FSSelMuxS = 0
                        FSSelMuxW = 0
                        FNSel = 0
                        FNMuxDirNS = 0
                        FNMuxDirEW = 0
                        FNSelMuxN = 0
                        FNSelMuxE = 0
                        FNSelMuxS = 0
                        FNSelMuxW = 0
                EndFU
        EndPath
```

```
EndProgram
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 2:

```
;
; Stream file created by COLTGUI, version code 841847636
; Created on Mon Sep 16 15:48:52 EDT 1996 by user bittner
;
#include "fielddef.h"
Path [0]
;
; Configuration for Port 2-1
;
       DataPort
              DPAddress = 25
              DPRWBit = 0
              DPSyncBit = 1
              DPSyncMP1 = 0
              DPSyncMP2 = 1
              DPSyncMP3 = 0
              DPSyncMP4 = 0
              DPSyncMP5 = 0
              DPSyncMP6 = 0
              DPLoopBit = 0
       EndDataPort
;
; Configuration for crossbar, Address = 9
;
       CrossBar
              Address = 9
              XBExtra = 0
       EndCrossBar
;
; Configuration for crossbar, Address = 8
;
       CrossBar
              Address = 8
              XBExtra = 0
       EndCrossBar
EndPath
EndProgram
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 3:

```
;
; Stream file created by COLTGUI, version code 841847636
; Created on Mon Sep 16 15:49:53 EDT 1996 by user bittner
```

```
;
#include "fielddef.h"
Path [0]
;
; Configuration for Port 3-1
;
        DataPort
                DPAddress = 26
                DPRWBit = 0
                DPSyncBit = 1
                DPSyncMP1 = 0
                DPSyncMP2 = 0
                DPSyncMP3 = 1
                DPSyncMP4 = 0
                DPSyncMP5 = 0
                DPSyncMP6 = 0
                DPLoopBit = 0
        EndDataPort
;
; Configuration for crossbar, Address = 3
;
        CrossBar
                Address = 3
                XBExtra = 0
        EndCrossBar
;
; Configuration for FU FU_33, Address = 33
;
        FU
                Address = 33
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 0
                ShiftCondInv = 1
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 0
                FSInv = 0
                CondMode = 0
                FNOutSel = 0
                LeftInputSelect = 0
                Lf = 0
                P = 12
                G = 0
                R = 12
                ProgramNorth = 0
                ProgramEast = 1
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 0
                DSelRight = 0
                DMuxDirNS = 0
```

385

```
                    DMuxDirEW = 1
                    DSelMuxN = 0
                    DSelMuxE = 3
                    DSelMuxS = 0
                    DSelMuxW = 0
                    FCSel = 0
                    FCMuxDirNS = 0
                    FCMuxDirEW = 0
                    FCSelMuxN = 0
                    FCSelMuxE = 0
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 0
                    FSMuxDirNS = 1
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 2
                    FSSelMuxW = 0
                    FNSel = 0
                    FNMuxDirNS = 0
                    FNMuxDirEW = 0
                    FNSelMuxN = 0
                    FNSelMuxE = 0
                    FNSelMuxS = 0
                    FNSelMuxW = 0
        EndFU
;
; Configuration for FU FU_37, Address = 37
;
        FU
                    Address = 37
                    SingleWordProgram = 0
                    ValidBitMux = 3
                    ShiftType = 0
                    ShiftCondInv = 0
                    ShiftCondInpSel = 0
                    LeftOperand = 131070
                    OutputDelay = 0
                    FCInpSel = 3
                    FCInv = 0
                    FNInpSel = 1
                    FNInv = 0
                    FSInpSel = 0
                    FSInv = 0
                    CondMode = 1
                    FNOutSel = 13
                    LeftInputSelect = 0
                    Lf = 3
                    P = 6
                    G = 8
                    R = 6
                    ProgramNorth = 0
                    ProgramEast = 0
                    ProgramSouth = 1
                    ProgramWest = 0
                    DSelLeft = 0
```

386

```
              DSelRight = 1
              DMuxDirNS = 1
              DMuxDirEW = 0
              DSelMuxN = 0
              DSelMuxE = 0
              DSelMuxS = 1
              DSelMuxW = 0
              FCSel = 6
              FCMuxDirNS = 0
              FCMuxDirEW = 0
              FCSelMuxN = 0
              FCSelMuxE = 0
              FCSelMuxS = 0
              FCSelMuxW = 0
              FSSel = 0
              FSMuxDirNS = 0
              FSMuxDirEW = 0
              FSSelMuxN = 0
              FSSelMuxE = 0
              FSSelMuxS = 0
              FSSelMuxW = 0
              FNSel = 6
              FNMuxDirNS = 0
              FNMuxDirEW = 0
              FNSelMuxN = 0
              FNSelMuxE = 0
              FNSelMuxS = 0
              FNSelMuxW = 0
      EndFU
;
; Configuration for FU FU_41, Address = 41
;
      FU
              Address = 41
              SingleWordProgram = 0
              ValidBitMux = 3
              ShiftType = 0
              ShiftCondInv = 0
              ShiftCondInpSel = 0
              LeftOperand = 0
              OutputDelay = 0
              FCInpSel = 3
              FCInv = 1
              FNInpSel = 1
              FNInv = 0
              FSInpSel = 1
              FSInv = 0
              CondMode = 0
              FNOutSel = 6
              LeftInputSelect = 0
              Lf = 0
              P = 15
              G = 0
              R = 0
              ProgramNorth = 0
              ProgramEast = 0
              ProgramSouth = 1
```

387

```
                    ProgramWest = 0
                    DSelLeft = 4
                    DSelRight = 4
                    DMuxDirNS = 1
                    DMuxDirEW = 1
                    DSelMuxN = 0
                    DSelMuxE = 3
                    DSelMuxS = 3
                    DSelMuxW = 0
                    FCSel = 0
                    FCMuxDirNS = 0
                    FCMuxDirEW = 0
                    FCSelMuxN = 0
                    FCSelMuxE = 0
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 0
                    FSMuxDirNS = 1
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 1
                    FSSelMuxW = 0
                    FNSel = 0
                    FNMuxDirNS = 0
                    FNMuxDirEW = 0
                    FNSelMuxN = 1
                    FNSelMuxE = 0
                    FNSelMuxS = 0
                    FNSelMuxW = 0
            EndFU
;
; Configuration for FU FU_45, Address = 45
;
        FU
                    Address = 45
                    SingleWordProgram = 0
                    ValidBitMux = 3
                    ShiftType = 0
                    ShiftCondInv = 0
                    ShiftCondInpSel = 0
                    LeftOperand = 98304
                    OutputDelay = 0
                    FCInpSel = 1
                    FCInv = 0
                    FNInpSel = 1
                    FNInv = 1
                    FSInpSel = 0
                    FSInv = 0
                    CondMode = 1
                    FNOutSel = 0
                    LeftInputSelect = 0
                    Lf = 3
                    P = 12
                    G = 0
                    R = 12
                    ProgramNorth = 0
```

```
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 4
                DSelRight = 4
                DMuxDirNS = 0
                DMuxDirEW = 0
                DSelMuxN = 0
                DSelMuxE = 0
                DSelMuxS = 0
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 1
                FSSel = 0
                FSMuxDirNS = 0
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 0
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 0
        EndFU
;
; Configuration for FU FU_34, Address = 34
;
        FU
                Address = 34
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 0
                ShiftCondInv = 0
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 0
                FSInv = 0
                CondMode = 0
                FNOutSel = 13
                LeftInputSelect = 0
                Lf = 0
                P = 6
                G = 8
```

389

```
                    R = 6
                    ProgramNorth = 0
                    ProgramEast = 0
                    ProgramSouth = 1
                    ProgramWest = 0
                    DSelLeft = 7
                    DSelRight = 3
                    DMuxDirNS = 0
                    DMuxDirEW = 0
                    DSelMuxN = 0
                    DSelMuxE = 0
                    DSelMuxS = 0
                    DSelMuxW = 0
                    FCSel = 0
                    FCMuxDirNS = 0
                    FCMuxDirEW = 1
                    FCSelMuxN = 0
                    FCSelMuxE = 2
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 0
                    FSMuxDirNS = 0
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 0
                    FSSelMuxW = 0
                    FNSel = 0
                    FNMuxDirNS = 0
                    FNMuxDirEW = 1
                    FNSelMuxN = 0
                    FNSelMuxE = 2
                    FNSelMuxS = 0
                    FNSelMuxW = 0
          EndFU
;
; Configuration for FU FU_38, Address = 38
;
          FU
                    Address = 38
                    SingleWordProgram = 0
                    ValidBitMux = 3
                    ShiftType = 0
                    ShiftCondInv = 0
                    ShiftCondInpSel = 0
                    LeftOperand = 98304
                    OutputDelay = 0
                    FCInpSel = 1
                    FCInv = 0
                    FNInpSel = 1
                    FNInv = 1
                    FSInpSel = 0
                    FSInv = 0
                    CondMode = 1
                    FNOutSel = 0
                    LeftInputSelect = 0
                    Lf = 3
```

```
                P - 12
                G = 0
                R = 12
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 0
                DSelRight = 0
                DMuxDirNS = 0
                DMuxDirEW = 0
                DSelMuxN = 0
                DSelMuxE = 0
                DSelMuxS = 0
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 1
                FSSel = 0
                FSMuxDirNS = 0
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 0
                FSSelMuxW = 0
                FNSel = 1
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 1
        EndFU
;
; Configuration for FU 1 Delay, Address = 42
;
        FU
                Address = 42
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 0
                ShiftCondInv = 0
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 0
                FSInv = 0
                CondMode = 1
                FNOutSel = 0
```

391

```
                LeftInputSelect = 0
                Lf = 0
                P = 0
                G = 0
                R = 0
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 7
                DSelRight = 7
                DMuxDirNS = 0
                DMuxDirEW = 0
                DSelMuxN = 0
                DSelMuxE = 0
                DSelMuxS = 0
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 0
                FSSel = 0
                FSMuxDirNS = 0
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 0
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 0
        EndFU
;
; Configuration for FU CombSignExp, Address = 46
;
        FU
                Address = 46
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 4
                ShiftCondInv = 1
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 3
                FSInv = 0
```

```
                CondMode - 0
                FNOutSel = 0
                LeftInputSelect = 0
                Lf = 0
                P = 12
                G = 0
                R = 12
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 3
                DSelRight = 0
                DMuxDirNS = 0
                DMuxDirEW = 1
                DSelMuxN = 0
                DSelMuxE = 3
                DSelMuxS = 0
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 0
                FSSel = 0
                FSMuxDirNS = 0
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 0
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 1
        EndFU
;
; Configuration for crossbar, Address = 15
;
        CrossBar
                Address = 15
                XBExtra = 0
        EndCrossBar
;
; Configuration for Port 5-1
;
        DataPort
                DPAddress = 28
                DPRWBit = 1
                DPSyncBit = 1
                DPSyncMP1 = 0
                DPSyncMP2 = 0
```

393

```
                    DPSyncMP3 = 0
                    DPSyncMP4 = 0
                    DPSyncMP5 = 0
                    DPSyncMP6 = 0
                    DPLoopBit = 0
            EndDataPort
EndPath
EndProgram
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 4:

```
;
; Stream file created by COLTGUI, version code 841847636
; Created on Mon Sep 16 15:50:47 EDT 1996 by user bittner
;
#include "fielddef.h"
Path [0]
;
; Configuration for Port 4-1
;
        DataPort
                DPAddress = 27
                DPRWBit = 0
                DPSyncBit = 1
                DPSyncMP1 = 0
                DPSyncMP2 = 0
                DPSyncMP3 = 0
                DPSyncMP4 = 1
                DPSyncMP5 = 0
                DPSyncMP6 = 0
                DPLoopBit = 0
        EndDataPort
;
; Configuration for crossbar, Address = 10
;
        CrossBar
                Address = 10
                XBExtra = 0
        EndCrossBar
;
; Configuration for crossbar, Address = 7
;
        CrossBar
                Address = 7
                XBExtra = 0
        EndCrossBar
;
; Configuration for FU FU_35, Address = 35
;
        FU
                Address = 35
                SingleWordProgram = 0
```

```
                ValidBitMux = 0
                ShiftType = 0
                ShiftCondInv = 1
                ShiftCondInpSel = 0
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 0
                FNInv = 0
                FSInpSel = 0
                FSInv = 0
                CondMode = 0
                FNOutSel = 0
                LeftInputSelect = 0
                Lf = 0
                P = 0
                G = 0
                R = 0
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 0
                DSelRight = 4
                DMuxDirNS = 1
                DMuxDirEW = 0
                DSelMuxN = 0
                DSelMuxE = 0
                DSelMuxS = 3
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 1
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 0
                FSSel = 0
                FSMuxDirNS = 1
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 2
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 1
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
                FNSelMuxW = 0
        EndFU
;
; Configuration for FU FU_39, Address = 39
;
        FU
```

```
                    Address = 39
                    SingleWordProgram = 0
                    ValidBitMux = 3
                    ShiftType = 0
                    ShiftCondInv = 0
                    ShiftCondInpSel = 0
                    LeftOperand = 0
                    OutputDelay = 0
                    FCInpSel = 3
                    FCInv = 1
                    FNInpSel = 1
                    FNInv = 0
                    FSInpSel = 1
                    FSInv = 0
                    CondMode = 0
                    FNOutSel = 6
                    LeftInputSelect = 0
                    Lf = 0
                    P = 15
                    G = 0
                    R = 0
                    ProgramNorth = 0
                    ProgramEast = 0
                    ProgramSouth = 1
                    ProgramWest = 0
                    DSelLeft = 4
                    DSelRight = 4
                    DMuxDirNS = 1
                    DMuxDirEW = 0
                    DSelMuxN = 0
                    DSelMuxE = 0
                    DSelMuxS = 3
                    DSelMuxW = 0
                    FCSel = 4
                    FCMuxDirNS = 0
                    FCMuxDirEW = 0
                    FCSelMuxN = 0
                    FCSelMuxE = 0
                    FCSelMuxS = 0
                    FCSelMuxW = 0
                    FSSel = 0
                    FSMuxDirNS = 1
                    FSMuxDirEW = 0
                    FSSelMuxN = 0
                    FSSelMuxE = 0
                    FSSelMuxS = 1
                    FSSelMuxW = 0
                    FNSel = 4
                    FNMuxDirNS = 0
                    FNMuxDirEW = 0
                    FNSelMuxN = 0
                    FNSelMuxE = 0
                    FNSelMuxS = 0
                    FNSelMuxW = 2
          EndFU
;
; Configuration for FU FU_43, Address = 43
```

```
;
        FU
              Address = 43
              SingleWordProgram = 0
              ValidBitMux = 0
              ShiftType = 0
              ShiftCondInv = 0
              ShiftCondInpSel = 0
              LeftOperand = 0
              OutputDelay = 0
              FCInpSel = 0
              FCInv = 0
              FNInpSel = 0
              FNInv = 0
              FSInpSel = 0
              FSInv = 0
              CondMode = 0
              FNOutSel = 0
              LeftInputSelect = 0
              Lf = 0
              P = 0
              G = 0
              R = 0
              ProgramNorth = 0
              ProgramEast = 0
              ProgramSouth = 1
              ProgramWest = 0
              DSelLeft = 0
              DSelRight = 0
              DMuxDirNS = 1
              DMuxDirEW = 0
              DSelMuxN = 0
              DSelMuxE = 0
              DSelMuxS = 3
              DSelMuxW = 0
              FCSel = 0
              FCMuxDirNS = 0
              FCMuxDirEW = 0
              FCSelMuxN = 0
              FCSelMuxE = 0
              FCSelMuxS = 0
              FCSelMuxW = 0
              FSSel = 0
              FSMuxDirNS = 1
              FSMuxDirEW = 0
              FSSelMuxN = 0
              FSSelMuxE = 0
              FSSelMuxS = 1
              FSSelMuxW = 0
              FNSel = 0
              FNMuxDirNS = 0
              FNMuxDirEW = 0
              FNSelMuxN = 0
              FNSelMuxE = 0
              FNSelMuxS = 0
              FNSelMuxW = 0
        EndFU
```

397

```
;
; Configuration for FU FU_47, Address = 47
;
        FU
                Address = 47
                SingleWordProgram = 0
                ValidBitMux = 3
                ShiftType = 0
                ShiftCondInv = 1
                ShiftCondInpSel = 1
                LeftOperand = 0
                OutputDelay = 0
                FCInpSel = 0
                FCInv = 0
                FNInpSel = 3
                FNInv = 0
                FSInpSel = 1
                FSInv = 1
                CondMode = 0
                FNOutSel = 10
                LeftInputSelect = 0
                Lf = 0
                P = 12
                G = 0
                R = 12
                ProgramNorth = 0
                ProgramEast = 0
                ProgramSouth = 1
                ProgramWest = 0
                DSelLeft = 4
                DSelRight = 4
                DMuxDirNS = 1
                DMuxDirEW = 0
                DSelMuxN = 0
                DSelMuxE = 0
                DSelMuxS = 1
                DSelMuxW = 0
                FCSel = 0
                FCMuxDirNS = 0
                FCMuxDirEW = 0
                FCSelMuxN = 0
                FCSelMuxE = 0
                FCSelMuxS = 0
                FCSelMuxW = 0
                FSSel = 4
                FSMuxDirNS = 0
                FSMuxDirEW = 0
                FSSelMuxN = 0
                FSSelMuxE = 0
                FSSelMuxS = 0
                FSSelMuxW = 0
                FNSel = 0
                FNMuxDirNS = 0
                FNMuxDirEW = 0
                FNSelMuxN = 0
                FNSelMuxE = 0
                FNSelMuxS = 0
```

398

```
              FNSelMuxW = 2
        EndFU
;
; Configuration for crossbar, Address = 16
;
        CrossBar
              Address = 16
              XBExtra = 0
        EndCrossBar
;
; Configuration for Port 6-1
;
        DataPort
              DPAddress = 29
              DPRWBit = 1
              DPSyncBit = 1
              DPSyncMP1 = 0
              DPSyncMP2 = 0
              DPSyncMP3 = 0
              DPSyncMP4 = 0
              DPSyncMP5 = 0
              DPSyncMP6 = 0
              DPLoopBit = 0
        EndDataPort
EndPath
EndProgram
```

The pwl input files giving the binary form of the stream headers are given below. The

following listing gives the pwl code for the stream header to be injected into *Data Port 1*:

```
# Initialize
# Path[0]
#     DataPort
LeftOperand = 1001000001011000
CLOCK
#     CrossBar
LeftOperand = 1000000000000001
CLOCK
#     FU
#         ConfigReg[0]
LeftOperand = 1000100011100000
CLOCK
#         ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#         ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#         ConfigReg[3]
LeftOperand = 0000000000000000
CLOCK
#         ConfigReg[4]
LeftOperand = 0000000000000000
```

399

```
CLOCK
#          ConfigReg[5]
LeftOperand = 0000110000001010
CLOCK
#          ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#          ConfigReg[7]
LeftOperand = 0000100001100000
CLOCK
#      FU
#          ConfigReg[0]
LeftOperand = 1000101001100100
CLOCK
#          ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#          ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#          ConfigReg[3]
LeftOperand = 0011000000000000
CLOCK
#          ConfigReg[4]
LeftOperand = 0001010000000000
CLOCK
#          ConfigReg[5]
LeftOperand = 0000110010000010
CLOCK
#          ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#          ConfigReg[7]
LeftOperand = 0000101110100101
CLOCK
#      FU
#          ConfigReg[0]
LeftOperand = 1000100011101000
CLOCK
#          ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#          ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#          ConfigReg[3]
LeftOperand = 0001000010000000
CLOCK
#          ConfigReg[4]
LeftOperand = 0000000000000100
CLOCK
#          ConfigReg[5]
LeftOperand = 0101100000001010
CLOCK
#          ConfigReg[6]
LeftOperand = 0011000000000000
CLOCK
```

400

```
#           ConfigReg[7]
LeftOperand = 0000000010100000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000001001101100
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[3]
LeftOperand = 0011000000000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0001010000000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0000110010000010
CLOCK
#           ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000101100000001
CLOCK
# End Programming Sequence
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 2:

```
# Initialize
# Path[0]
#       DataPort
LeftOperand = 1001000010011001
CLOCK
#       CrossBar
LeftOperand = 1000000000001001
CLOCK
#       CrossBar
LeftOperand = 1000000000001000
CLOCK
# End Programming Sequence
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 3:

```
# Initialize
# Path[0]
#       DataPort
LeftOperand = 1001000100011010
CLOCK
#       CrossBar
LeftOperand = 1000000000000011
CLOCK
#       FU
#             ConfigReg[0]
LeftOperand = 1001100011100001
CLOCK
#             ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#             ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#             ConfigReg[3]
LeftOperand = 0000000000000000
CLOCK
#             ConfigReg[4]
LeftOperand = 0000000000000000
CLOCK
#             ConfigReg[5]
LeftOperand = 0000110000001110
CLOCK
#             ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#             ConfigReg[7]
LeftOperand = 0000100001100000
CLOCK
#       FU
#             ConfigReg[0]
LeftOperand = 1000101111100101
CLOCK
#             ConfigReg[1]
LeftOperand = 0111111111111110
CLOCK
#             ConfigReg[2]
LeftOperand = 0000000000110010
CLOCK
#             ConfigReg[3]
LeftOperand = 0100101000000000
CLOCK
#             ConfigReg[4]
LeftOperand = 0110001000011100
CLOCK
#             ConfigReg[5]
LeftOperand = 0101101010000000
CLOCK
#             ConfigReg[6]
LeftOperand = 0011000000000000
CLOCK
#             ConfigReg[7]
LeftOperand = 0000001100000000
```

402

```
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100011101001
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0000010000001000
CLOCK
#           ConfigReg[3]
LeftOperand = 0100011100000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0100011000000110
CLOCK
#           ConfigReg[5]
LeftOperand = 0000000110001110
CLOCK
#           ConfigReg[6]
LeftOperand = 0111100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000000010100000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000101111101101
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0000000000001000
CLOCK
#           ConfigReg[3]
LeftOperand = 0110100100001000
CLOCK
#           ConfigReg[4]
LeftOperand = 0000001000000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0000110000000000
CLOCK
#           ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000001100000000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100011100010
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
```

403

```
CLOCK
#           ConfigReg[2]
LeftOperand = 0000000010000111
CLOCK
#           ConfigReg[3]
LeftOperand = 0000000110000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0110000010000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0101101001100000
CLOCK
#           ConfigReg[6]
LeftOperand = 0011000000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000000000000000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000101111100110
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0100000001000000
CLOCK
#           ConfigReg[3]
LeftOperand = 0110100000001000
CLOCK
#           ConfigReg[4]
LeftOperand = 0000001000000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0000110000000000
CLOCK
#           ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000001100000000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100011101010
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0000000000001111
CLOCK
#           ConfigReg[3]
LeftOperand = 0000100110000000
CLOCK
```

```
#           ConfigReg[4]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[6]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000000000000000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100011101110
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0100000000000001
CLOCK
#           ConfigReg[3]
LeftOperand = 0000000010000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0000110000000000
CLOCK
#           ConfigReg[5]
LeftOperand = 0000110000001110
CLOCK
#           ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0100100000000000
CLOCK
#       CrossBar
LeftOperand = 1000000000001111
CLOCK
#       DataPort
LeftOperand = 1101000000011100
CLOCK
# End Programming Sequence
```

The following listing gives the dfc code for the stream header to be injected into *Data*

Port 4:

```
# Initialize
# Path[0]
#       DataPort
LeftOperand = 1001001000011011
CLOCK
```

```
#       CrossBar
LeftOperand - 1000000000001010
CLOCK
#       CrossBar
LeftOperand = 1000000000000111
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100000100011
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0000000000001000
CLOCK
#           ConfigReg[3]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0000000001000111
CLOCK
#           ConfigReg[5]
LeftOperand = 0000000010000000
CLOCK
#           ConfigReg[6]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000100001100000
CLOCK
#       FU
#           ConfigReg[0]
LeftOperand = 1000100011100111
CLOCK
#           ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#           ConfigReg[2]
LeftOperand = 0010000000011000
CLOCK
#           ConfigReg[3]
LeftOperand = 0100011100000000
CLOCK
#           ConfigReg[4]
LeftOperand = 0100011000001110
CLOCK
#           ConfigReg[5]
LeftOperand = 0000000110000000
CLOCK
#           ConfigReg[6]
LeftOperand = 0111100000000000
CLOCK
#           ConfigReg[7]
LeftOperand = 0000000010100000
CLOCK
#       FU
```

406

```
#            ConfigReg[0]
LeftOperand = 1000100000101011
CLOCK
#            ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#            ConfigReg[2]
LeftOperand = 0000000000000000
CLOCK
#            ConfigReg[3]
LeftOperand = 0000000000000000
CLOCK
#            ConfigReg[4]
LeftOperand = 0000000000000110
CLOCK
#            ConfigReg[5]
LeftOperand = 0000000010000000
CLOCK
#            ConfigReg[6]
LeftOperand = 0000000000000000
CLOCK
#            ConfigReg[7]
LeftOperand = 0000000010100000
CLOCK
#       FU
#            ConfigReg[0]
LeftOperand = 1000100011101111
CLOCK
#            ConfigReg[1]
LeftOperand = 0000000000000000
CLOCK
#            ConfigReg[2]
LeftOperand = 0010000000001000
CLOCK
#            ConfigReg[3]
LeftOperand = 0000000100000000
CLOCK
#            ConfigReg[4]
LeftOperand = 0011011100000100
CLOCK
#            ConfigReg[5]
LeftOperand = 0000110110000000
CLOCK
#            ConfigReg[6]
LeftOperand = 0001100000000000
CLOCK
#            ConfigReg[7]
LeftOperand = 0000110000000001
CLOCK
#       CrossBar
LeftOperand = 1000000000010000
CLOCK
#       DataPort
LeftOperand = 1101000000011101
CLOCK
```

```
# End Programming Sequence
```

These binary files can be directly injected into Colt to configure the floating point multiplier. For simulation purposes using Cadence, it may be necessary to create an STL file that contains all four stream headers, each feeding into its corresponding port. Ray Bittner wrote a utility to accomplish this called mergestl. This program takes six pwl files, one for each data port, as input and produces a merged STL file that is suitable for simulation. If a given port is used as an output, then a file containing the word "Output" on the first line should be substituted into the mergestl command line specification. The STL simulation file for the floating point multiplier is given below. Note that header and trailer information has been added to the file to define pins, clock rate, etc. The listing below gives the STL floating point multiplier simulation file.

```
; ---STL Source Program Template---

stlinit
;settarget spice

; note that clock inputs should be of type "clk"
defpin Clock            clk
defpin Port0ProgramPin in
defpin Port0ReceivePin in
defpin Port0TransmitPin         in
defpin Port0WritePin    in
defpin Port1ProgramPin in
defpin Port1ReceivePin in
defpin Port1TransmitPin         in
defpin Port1WritePin    in
defpin Port2ProgramPin in
defpin Port2ReceivePin in
defpin Port2TransmitPin         in
defpin Port2WritePin    in
defpin Port3ProgramPin in
defpin Port3ReceivePin in
defpin Port3TransmitPin         in
defpin Port3WritePin    in
defpin Port4ProgramPin in
defpin Port4ReceivePin in
defpin Port4TransmitPin         in
defpin Port4WritePin    in
defpin Port5ProgramPin in
defpin Port5ReceivePin in
defpin Port5TransmitPin         in
defpin Port5WritePin    in
defpin Reset            in
defpin Port0PinBus<15:0>        io
defpin Port1PinBus<15:0>        io
```

```
defpin Port2PinBus<15:0>        io
defpin Port3PinBus<15:0>        io
defpin Port4PinBus<15:0>        io
defpin Port5PinBus<15:0>        io


deftiming 1e-10 5e-09 4e-08  ; define basic time units
; define any clocks or timing (defclock, defstrobe)
defclock "....1111" Clock

; define a basic format statement consisting of all the
; input and bidirectional nodes
defformat Reset \
        Port0ProgramPin Port0WritePin Port0TransmitPin Port0ReceivePin Port0PinBus \
        Port1ProgramPin Port1WritePin Port1TransmitPin Port1ReceivePin Port1PinBus \
        Port2ProgramPin Port2WritePin Port2TransmitPin Port2ReceivePin Port2PinBus \
        Port3ProgramPin Port3WritePin Port3TransmitPin Port3ReceivePin Port3PinBus \
        Port4ProgramPin Port4WritePin Port4TransmitPin Port4ReceivePin Port4PinBus \
        Port5ProgramPin Port5WritePin Port5TransmitPin Port5ReceivePin Port5PinBus


deftest
;stlprintf(spice "vdd [#VDD!] 0 5\n")
;stlprintf(spice "vgnd [#GND!] 0 0\n")


;reset
xv(0    1 1 1 1 Z 0              1 1 1 1 Z 0 \
        1 1 1 1 Z 0              1 1 1 1 Z 0 \
        1 1 1 1 Z 0              1 1 1 1 Z 0 )


;begin programming data ports

xv(1 0 0 1 1 0b1001000001011000 X    0 0 1 1 0b1001000010011001 X
     0 0 1 1 0b1001000100011010 X    0 0 1 1 0b1001001001000011011 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b1000000000000001 X    0 0 1 1 0b1000000000001001 X
     0 0 1 1 0b1000000000000011 X    0 0 1 1 0b1000000000001010 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b1000100011100000 X    0 0 1 1 0b1000000000001000 X
     0 0 1 1 0b1001100011100001 X    0 0 1 1 0b1000000000000111 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0000000000000000 X    0 0 1 1 0b1000100000100011 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0000000000000000 X    0 0 1 1 0b0000000000000000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0000000000000000 X    0 0 1 1 0b0000000000001000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0000000000000000 X    0 0 1 1 0b0000000000000000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000110000001010 X    1 1 0 1 0 X
     0 0 1 1 0b0000110000001110 X    0 0 1 1 0b0000000001000111 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0001100000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0001100000000000 X    0 0 1 1 0b0000000010000000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000100001100000 X    1 1 0 1 0 X
     0 0 1 1 0b0000100001100000 X    0 0 1 1 0b0000000000000000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b1000101001100100 X    1 1 0 1 0 X
     0 0 1 1 0b1000101111100101 X    0 0 1 1 0b0000100001100000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0111111111111110 X    0 0 1 1 0b1000100011100111 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0000000000000000 X    1 1 0 1 0 X
     0 0 1 1 0b0000000000110010 X    0 0 1 1 0b0000000000000000 X
     1 1 1 1 Z 0                      1 1 1 1 Z 0                   )
xv(1 0 0 1 1 0b0011000000000000 X    1 1 0 1 0 X
```

409

```
       0  0  1  1  0b0100101000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0001010000000000 X
       0  0  1  1  0b0110001000011100 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000011001000010 X
       0  0  1  1  0b0101101010000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0001100000000000 X
       0  0  1  1  0b0001100000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000101110100101 X
       0  0  1  1  0b0000000110000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b1000100011101000 X
       0  0  1  1  0b1000100011101001 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000000000000 X
       0  0  1  1  0b0000000000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000000000000 X
       0  0  1  1  0b0000001000001000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0001000010000000 X
       0  0  1  1  0b0100011100000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000000000100 X
       0  0  1  1  0b0100011000000110 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0101100000001010 X
       0  0  1  1  0b0000000110001110 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0011000000000000 X
       0  0  1  1  0b0111100000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000010100000 X
       0  0  1  1  0b0000000010100000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b1000001001101100 X
       0  0  1  1  0b1000101111101101 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000000000000 X
       0  0  1  1  0b0000000000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000000000000000 X
       0  0  1  1  0b0000000000001000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0011000000000000 X
       0  0  1  1  0b0110100100001000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0001010000000000 X
       0  0  1  1  0b0000001000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000110010000010 X
       0  0  1  1  0b0000110000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0001100000000000 X
       0  0  1  1  0b0001100000000000 X
       1  1  1  1  Z  0
xv(1   0  0  1  1  0b0000101100000001 X
       0  0  1  1  0b0000000110000000 X
       1  1  1  1  Z  0
xv(1   1  1  0  1  0  X
       0  0  1  1  0b1000100011100010 X
       1  1  1  1  Z  0
xv(1   1  1  0  1  0  X
       0  0  1  1  0b0000000000000000 X
       1  1  1  1  Z  0
xv(1   1  1  0  1  0  X
       0  0  1  1  0b0000000000010000111 X
       1  1  1  1  Z  0

0  0  1  1  0b0010000000011000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0100011100000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0100011000001110 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000110000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0111100000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000010100000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b1000100000101011 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000110 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000010000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000010100000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b1000100011101111 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0010000000001000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000000100000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0011011100000100 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000110110000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0001100000000000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b0000110000000001 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b1000000000010000 X
1  1  1  1  Z  0                              )
1  1  0  1  0  X
0  0  1  1  0b1101000000011101 X
1  1  1  1  Z  0                              )
```

```
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b00000000110000000 X      1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0110000010000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0101101001100000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0011000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b1000101111100110 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0100000001000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0110100000001000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000001000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000110000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0001100000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000001100000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b1000100011101010 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000001111 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000100110000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b1000100011101110 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0000000000000000 X        1 1 0 1 0 X
       1 1 1 1 Z 0                         1 1 1 1 Z 0                                    )
xv(1 1 1 0 1 0 X                          1 1 0 1 0 X
       0 0 1 1 0b0100000000000001 X        1 1 0 1 0 X
```

411

```
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b0000000010000000 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b0000110000000000 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b0000110000001110 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b0001100000000000 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b0100100000000000 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b1000000000001111 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        0 0 1 1 0b1101000000011100 X               1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
xv(1 1 1 0 1 0 X                                   1 1 0 1 0 X
        1 1 0 1 0 X                                1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )

;Put in one more blank to avoid skip bus jump ahead problem
xv(1    1 1 0 1 0 X                                1 1 0 1 0 X
        1 1 0 1 0 X                                1 1 0 1 0 X
        1 1 1 1 Z 0                                1 1 1 1 Z 0                              )
;send in data
xv(1    1 1 1 1 0x3452 X                           1 1 1 1 0xFEB0 X \
        1 1 1 1 0xDA09 X                           1 1 1 1 0xDE1A X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 0 1 0xE4EE X                           1 1 1 1 0xF10A X \
        1 1 1 1 0x1DBE X                           1 1 1 1 0xC00D X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0xE4EE X                           1 1 1 1 0xF10A X \
        1 1 1 1 0x1DBE X                           1 1 1 1 0xC00D X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0xE4EE X                           1 1 1 1 0xF10A X \
        1 1 1 1 0x1DBE X                           1 1 1 1 0xC00D X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0xE4EE X                           1 1 1 1 0xF10A X \
        1 1 1 1 0x1DBE X                           1 1 1 1 0xC00D X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0x0001 X                           1 1 1 1 0x8000 X \
        1 1 1 1 0xFFFF X                           1 1 1 1 0x8000 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0x0001 X                           1 1 1 1 0x8000 X \
        1 1 1 1 0x01FF X                           1 1 1 1 0x8000 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0x0001 X                           1 1 1 1 0x8000 X \
        1 1 1 1 0x81FF X                           1 1 1 1 0x8000 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 1 1 0x8001 X                           1 1 1 1 0x8000 X \
        1 1 1 1 0x01FF X                           1 1 1 1 0x8000 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )

; send dummy data to push this through the pipeline
xv(1    1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 1 1 Z 0                                1 1 1 1 Z 0 )
xv(1    1 1 0 1 0 X                                1 1 0 1 0 X \
        1 1 0 1 0 X                                1 1 0 1 0 X \
```

```
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )
    xv(1        1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 0 1 0 X                    1 1 0 1 0 X \
                1 1 1 1 Z 0                    1 1 1 1 Z 0 )

    endtest
```

Finally, pwl is another program that was written by Ray Bittner to aid simulation. Pwl will accept a PWL file as input and will output either a SPICE, STL or VHDL stimulus file. This is useful for simulating a single stream using one of these simulation tools. A pwl.cfg file must be specified that gives timing information to be used in the output, an example of this file is:

```
ClockPeriod = 40e-9
ClockALowTime = 16e-9
ClockBHighTime = 13e-9
ClockAStartDelay = 0.0
ClockBStartDelay = 2e-9
EdgeRiseTime = 100e-12
EdgeFallTime = 100e-12
```

The parameters in the file are shown above. The program generates clocking for the twin clock flip flops that are used throughout the design; thus the user controls the overall clock period of the output waveforms, the amount of time that *ClockA* should be low, the amount of time that *ClockB* should be high and the delay from the "beginning" of the clock period until each phase should rise. The rise and fall times for all signals are given last. The program also

expects to find header.vhd, trailer.vhd, header.stl and trailer.stl files in the directory in which it is run. These are used to automatically insert preamble and epilogue code into the output file and can be empty, if desired.

Vita

I was born in the small town of Frostburg, Maryland on February 5, 1969. However, most of my childhood was spent in Ellerslie, Maryland, where I attended my first three years of school in a three room school house. After finishing the sixth grade in Cash Valley Elementary, I entered Mount Savage High School where I received my diploma as one of 63 graduating students. From the summer of my junior year well into my college carreer at Virginia Tech, I worked at McDonald's and for the Allegany County Department of Public Works. After four years as an undergraduate at Virginia Tech, I was conferred a Bachelor of Science in Computer Engineering, with a minor in Computer Science in May of 1991. The following summer I was employed by IBM in Poughkeepsie, New York. Continuing at Tech, I received a Master of Science in Electrical Engineering in May of 1993 for my work with neural networks, which included a deterministic generation algorithm and VLSI implementation. During the summers of '93 and '94 I worked as a software intern with Microsoft in Redmond, Washington. After my master's, I was offered a Bradley Fellowship with the Viriginia Tech Electrical Engineering department, which persuaded me to continue at Tech for my doctoral work. I persued this degree through January of 1997, when I defended this thesis. Nine and a half years after I first set foot on the Virginia Tech campus, I left it a changed man; older, more knowledgeable and, hopefully, wiser.

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

❑ **BLACK BORDERS**

❑ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**

❑ **FADED TEXT OR DRAWING**

☑ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**

❑ **SKEWED/SLANTED IMAGES**

❑ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**

❑ **GRAY SCALE DOCUMENTS** .

❑ **LINES OR MARKS ON ORIGINAL DOCUMENT**

❑ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**

❑ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.